2/5 NASA-CR-202191

public avail

DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

NASA

REVIEW OF ICLASS

OCTOBER 18, 1991

INFORMATION SCIENCES LIBRARY
AMES RESEARCH CENTER
MOFFETT FIELD, CALIF.

DEC 19 1991

Viewgraphs and Reprints on

Software Systems, Architectures and Tools

Software Engineering and Artificial Intelligence

INFORMATION SCIENCES LIBRARY
AMES RESEARCH CENTER
MOFFETT FIELD, CALIF.
DEC 19 1991

NASA ANNUAL REVIEW

Illinois Computing Laboratory for Aerospace System and Software (ICLASS)

Room 1310, Conference Room, Digital Computing Laboratory
University of Illinois at Urbana-Champaign
October 18, 1991

7:45 - 8:15	Coffee and Pastries	
8:15 - 8:25	Welcome	
8:25 - 8:35	Overview on Distributed and Real-Time Systems Software Engineering and AI	J. Liu
8:35 - 8:55	Reliable, Distributed Data Base Management	G. Belford
8:55 - 9:15	Scheduling Algorithms for Replicated Real-Time Tasks	K-J. Lin
9:15 - 9:35	A Prototyping Environment for Real-Time Systems	J. Liu
9:35 - 9:55	GST: Grain-size Tuning for Efficient Execution	A. Chien
9:55 - 10:15	Scalable Performance Environments for Parallel Systems	D. Reed
10:15 - 10:45	Coffee break, Demonstrations and Posters	
10:45 - 11:05	Multiprocessor Operating Systems	R. Campbell
11:05 - 11:25	Constraint Processing in Logic Programming and Automated Deduction	A. Frisch
11:25 - 11:45	Spectral Methods for Meteorology on Advanced Computer Architectures	F. Saied
11:45 - 12:05	Comparing Languages for Maintainability	S. Kamin
12:05 - 12:25	Schema-based and Case-based Software Design	M. Harandi
12:25 - 1:30	Lunch (2240 DCL)	
1:30 - 1:45	Overview on Parallel Architectures and Algorithms and Reliable, Fault-Tolerant Computing	R. Iyer
1:45 - 2:10	Mission Adaptive Multiprocessor Architectures	J. Patel
2:10 - 2:35	TEACHER: A System for Automatic Creation of Heuristics	B. Wah
2:35 - 3:00	IMPACT: Compilation for High-Performance Parallel Systems	W-M. Hwu
3:00 - 3:30	Coffee Break, Demonstrations and Posters	
3:30 - 3:55	Parallel Algorithms for Computer-Aided Design	P. Bannerjee
3:55 - 4:20	Design for High Performance and Dependability	R. Iyer
4:20 - 4:45	Recovery in Dependable Parallel Architectures	K. Fuchs
4:45 - 5:10	Compiled Switch-Level Simulation	D. Saab

Presentations on

Software Systems, Architectures and Tools; Software Engineering and Artificial Intelligence

•	Reliable, Distributed Data Base Management	G. Belford
o	Scheduling Replicated Imprecise Computations	K.-J. Lin (A. Yu)
o	PERTS: A Prototyping Environment for Real-Time Systems	J. Liu
*	GST: Grain-Size Tuning for Efficient Execution	A. Chien
•	Scalable Performance Environments for Parallel Systems	D. Reed
•	Multiprocessor Operating Systems	R. Campbell (L. L. Yuen)
•	Constraint Processing in Logic Programming and Automated Deduction	A. Frisch
*	Spectral Methods for Meteorology on Advanced Computer Architectures	F. Saied
o	Comparing Languages for Maintainability	S. Kamin
•	Schema-Based and Case-Based Software Design	M. Harandi

• — continuing projects; * — new projects

o — continuing projects in new directions

Prototypes Completed

- APPROXIMATE: a query processor that returns monotonically improving answers to relational algebra queries

- FLEX: a programming environment for writing flexible real-time software

- IMMORTAL: a prototype belief revision system

Posters

- Anti-unification with Taxonomic Information — *C. David Page*
- Performance Tools for Parallel Systems — *Tara Madhyastha and Bradley Schwartz*
- Design Schema Acquisition — *Hing-Yan Lee*
- Convex Ceiling Protocol: A Concurrency Control Algorithm for Real-Time Systems — *Hidenori Nakazato*
- Scheduling Replicated Real-Time Tasks for Fault Tolerance — *Albert C. Yu*

Viewgraphs

OCTOBER 18, 1991

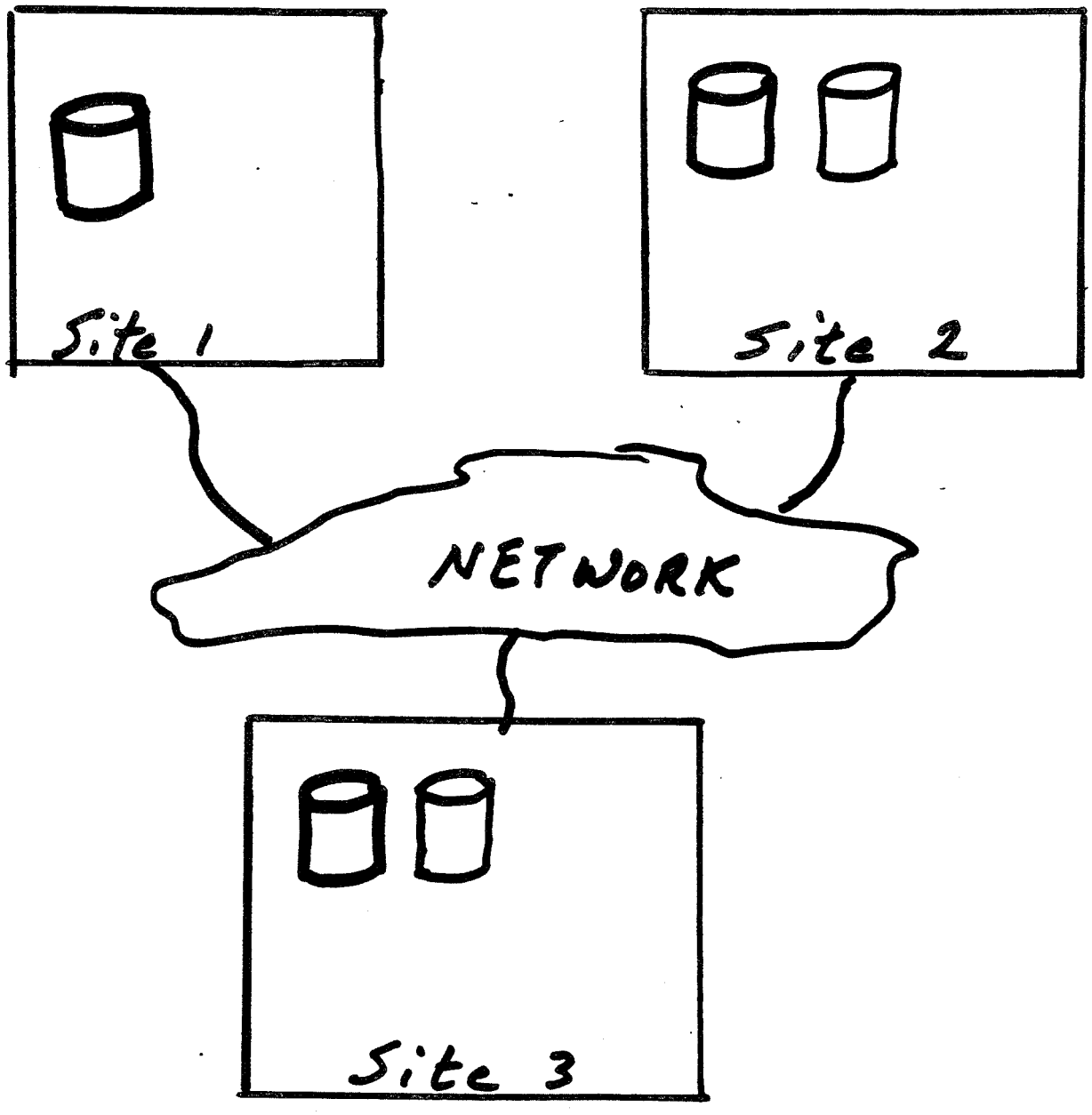
EFFICIENT, RELIABLE DISTRIBUTED DATABASES

GENEVA G. BELFORD

PROFESSOR OF COMPUTER SCIENCE

October 18, 1991

Distributed Data



Data may be fully, partially, or non replicated.

What is a DATABASE TRANSACTION?

Sequence of READs and WRITEs of data items

Bracketed by BEGIN, END

Possible computations between Reads and Writes

DISTRIBUTED TRANSACTION

Data at more than one site accessed

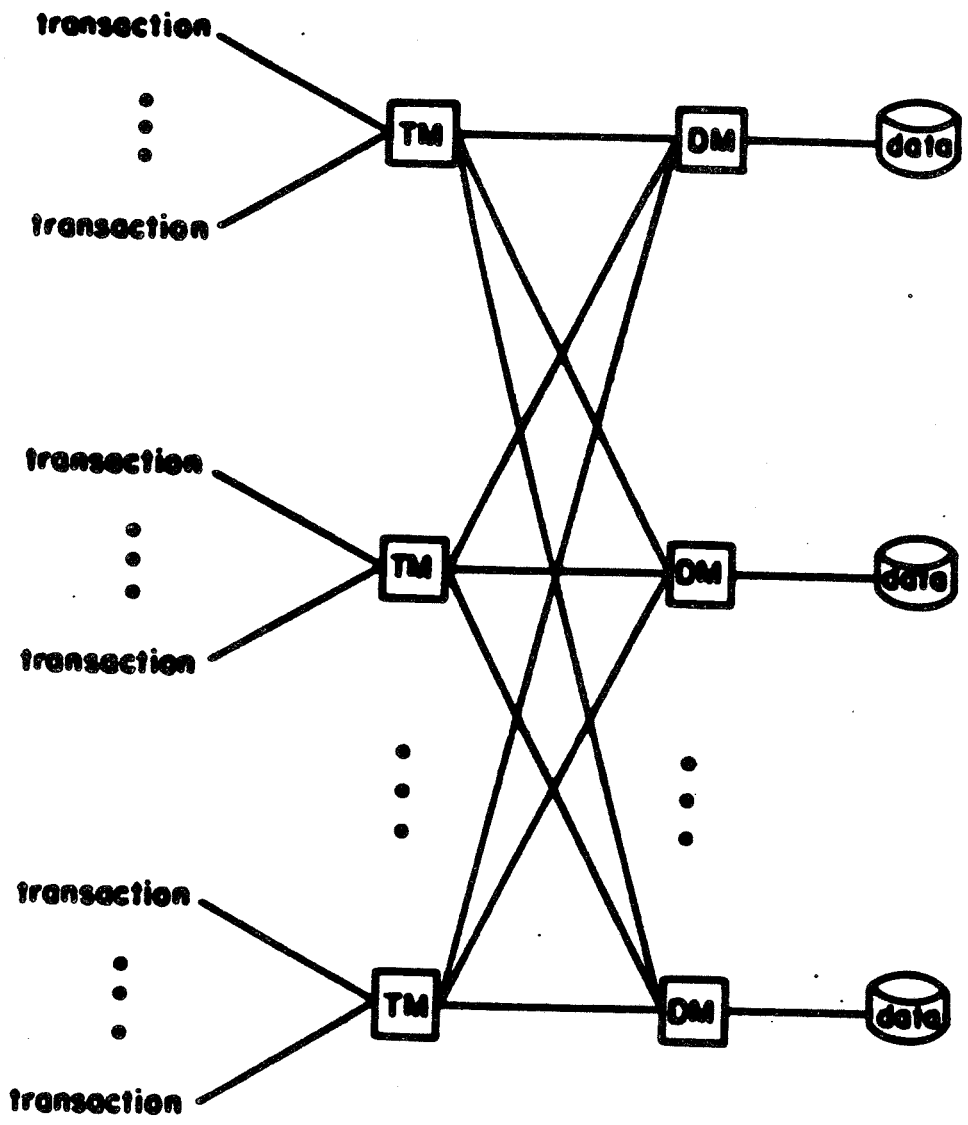
If data are replicated:

READ *ONE* COPY

WRITE *ALL* COPIES

Models for Distributed Transaction Processing

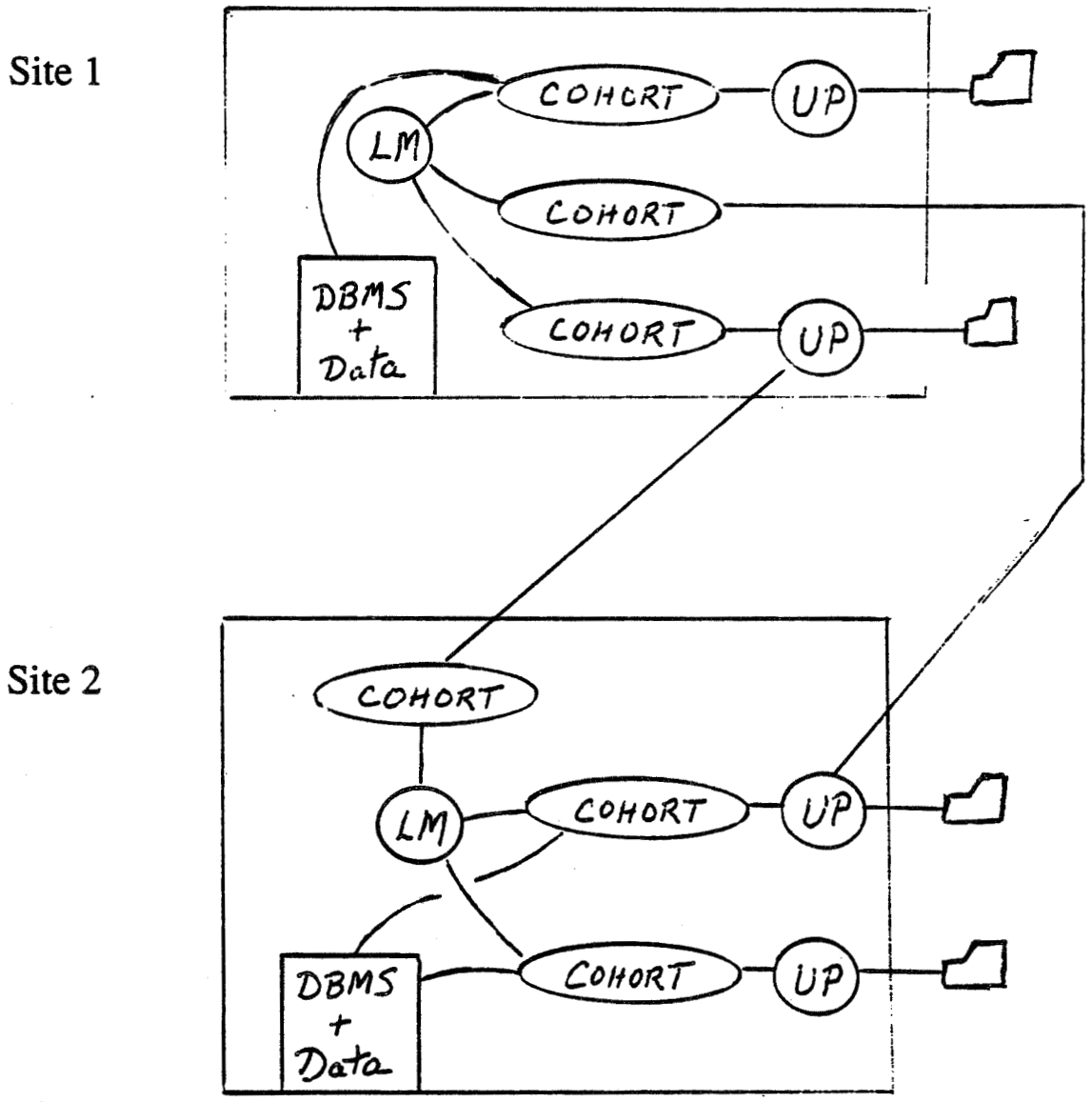
Classic model:



DM, TM overburdened?
MODULARIZE!

THE DTROLL DISTRIBUTED DATABASE SYSTEM

SYSTEM ARCHITECTURE



UP = User Process; LM = Lock Manager

RELIABILITY

WHAT CAN GO WRONG?

Network Failures

Site Failures

Logs and Recovery Procedures Help --

BUT may be inadequate if TM/UP (or its site) fails.

RELIABILITY THROUGH REPLICATION

REPLICATE TRANSACTIONS!

I.E., TM and Cohorts replicated on different sites.

ACTIVE VS. PASSIVE REPLICAS

ACTIVE REPLICAS

Entire transactions replicated at different sites

(Assumes fully replicated data)

Replicas run in parallel.

Fastest replica to complete execution returns results.

Other replicas (including failed ones) ignored.

Problems of "Inconsistent Execution"

[Ref. S. Shi, Ph.D. thesis, 1990]

PASSIVE REPLICAS

"Backup copies" of modules

Backups are kept informed by "Primary"

If Primary fails, Backup (eventually) takes over.

A new approach to passive replication.

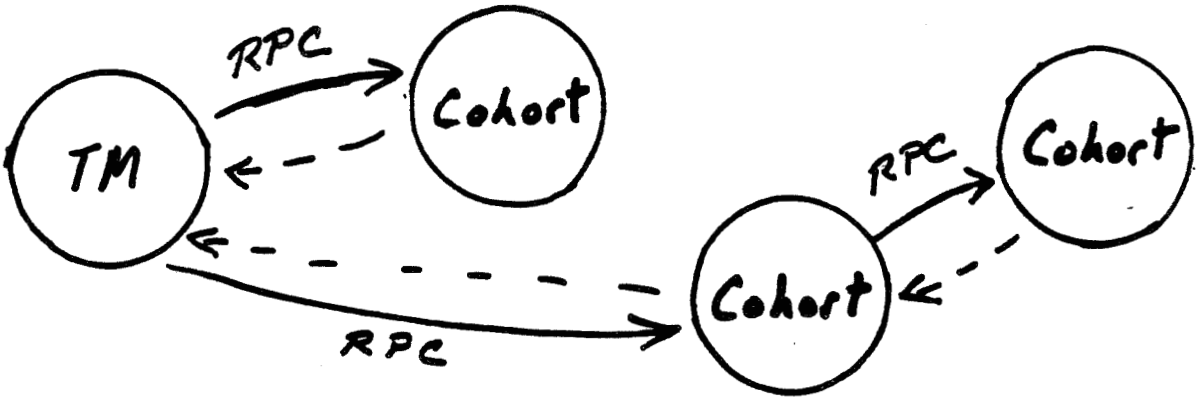
MIGRATING CONTROL OF DATABASE TRANSACTIONS

[David Wright, RA (NASA), Jan. '90 —]

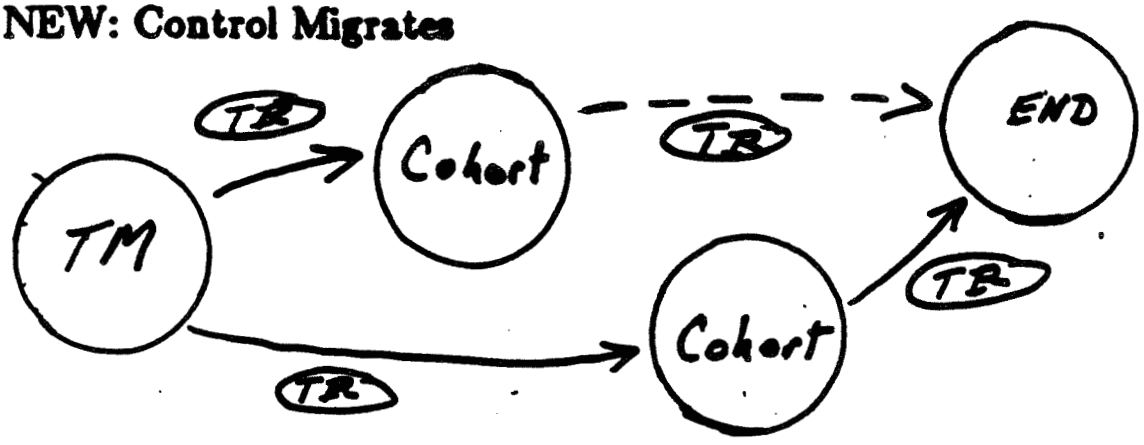
→ **A New Paradigm for Enhanced Recoverability**

AND Dynamic Decisions

- **OLD: Transaction Controlled from 'Root'**



- **NEW: Control Migrates**



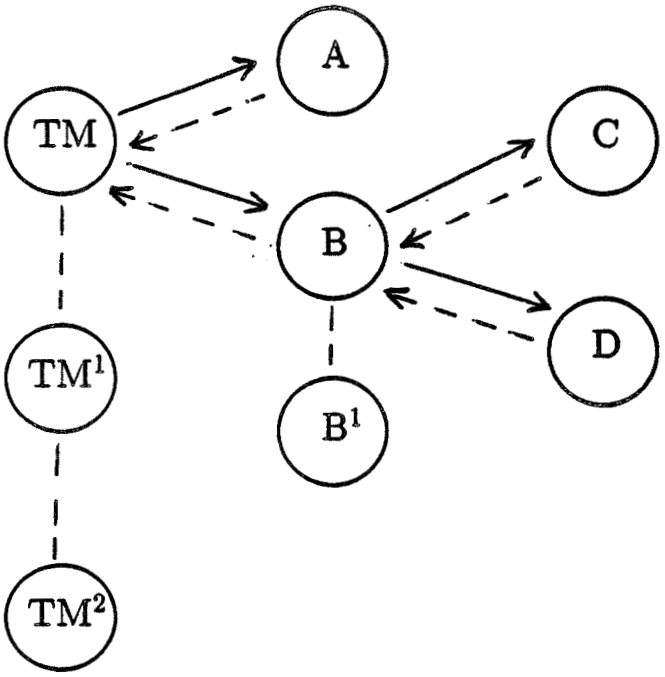
TM = Transaction Manager/Initiator

TR= Transaction Record

RPC = Remote Procedure Call

Recovery & Migrating Control

RPC: Replicate critical modules

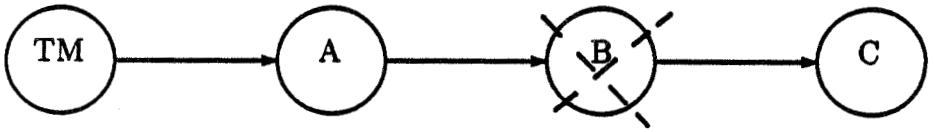


Control of transaction, and
for recovery, resides at TM.

With Migrating Control Scheme:

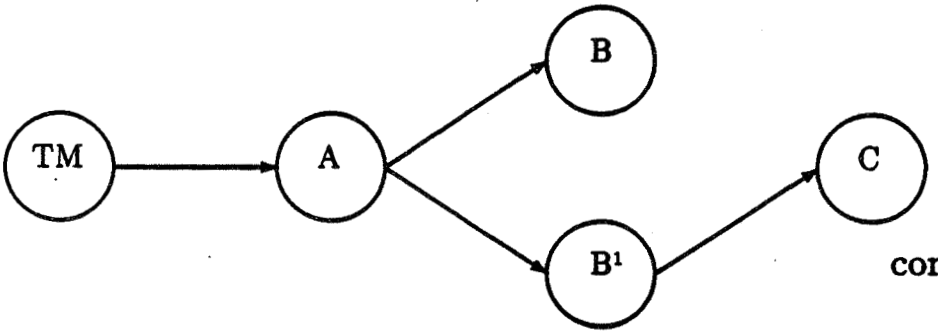
* Active Site

Suppose B fails, leaving C "isolated":



* becomes active, creates new 'B'

isolated



complete!

* Replicas only activated when needed.

* No need to back up to TM.

IN MORE DETAIL:

COMPILER ANALYZES TRANSACTION

Transaction COMPONENTS, each separately mappable to a site.

FEASIBLE sites for component execution

(data available)

Transaction STRUCTURE -- including parallelism

and alternative strategies.

RESULTING TRANSACTION 'IMAGE' DISTRIBUTED TO SITES

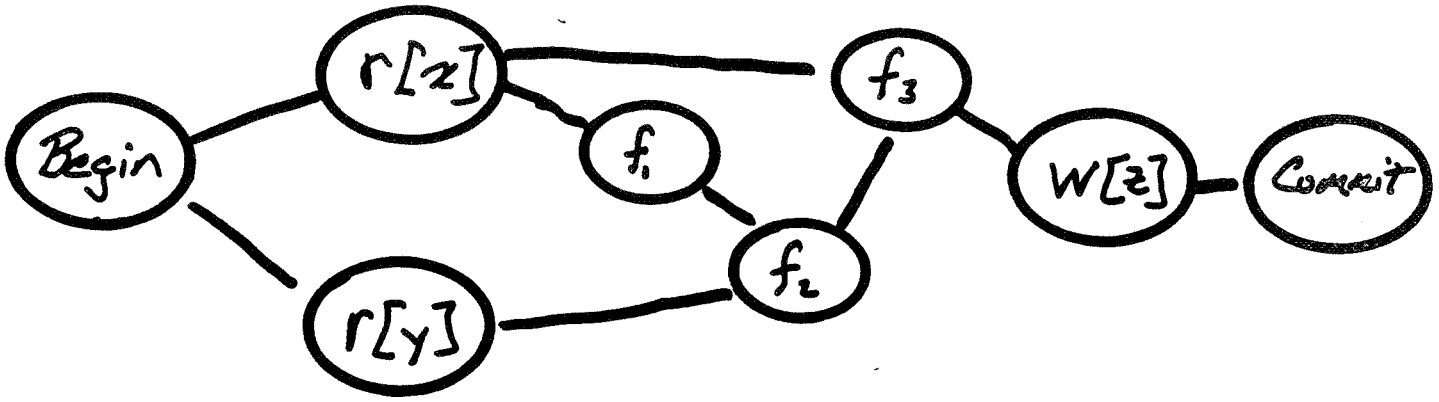
(All available sites where a component may run)

AFTER TRANSACTION INITIATION:

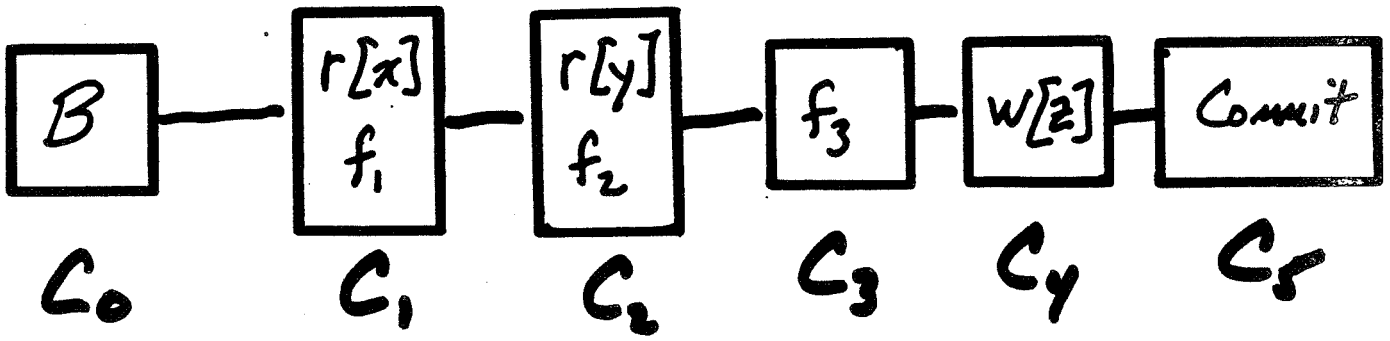
Components DYNAMICALLY decide where successor(s) run.

EXECUTE Messages sent with Control, History info.

E.G., A Transaction Specification:



is transformed by the compiler into a Decision Graph:



with 6 components C_i.

Note: "Transient" Input/Output links components.

E.g. C_1 has two outputs -

L_1 (result of read) $\rightarrow C_3$

L_2 (result of f_1) $\rightarrow C_2$

Component Code
+
Tabular representation of
Trans structure, I/O

Comp	In	Out	T. in	T. out	Preds	Succs
C_0						C_1
C_1	x			L_1, L_2	C_0	C_2
C_2	y		L_2	L_3	C_1	C_3
C_3			L_1, L_3	L_4	C_2	C_4
C_4		z	L_4		C_3	C_5
C_5					C_4	

The Transaction Image

- Distributed around network
- Note assumption -
  Predefined transactions
  run often enough to make
  this worth while.

Suppose data items x, y, z are located as follows:

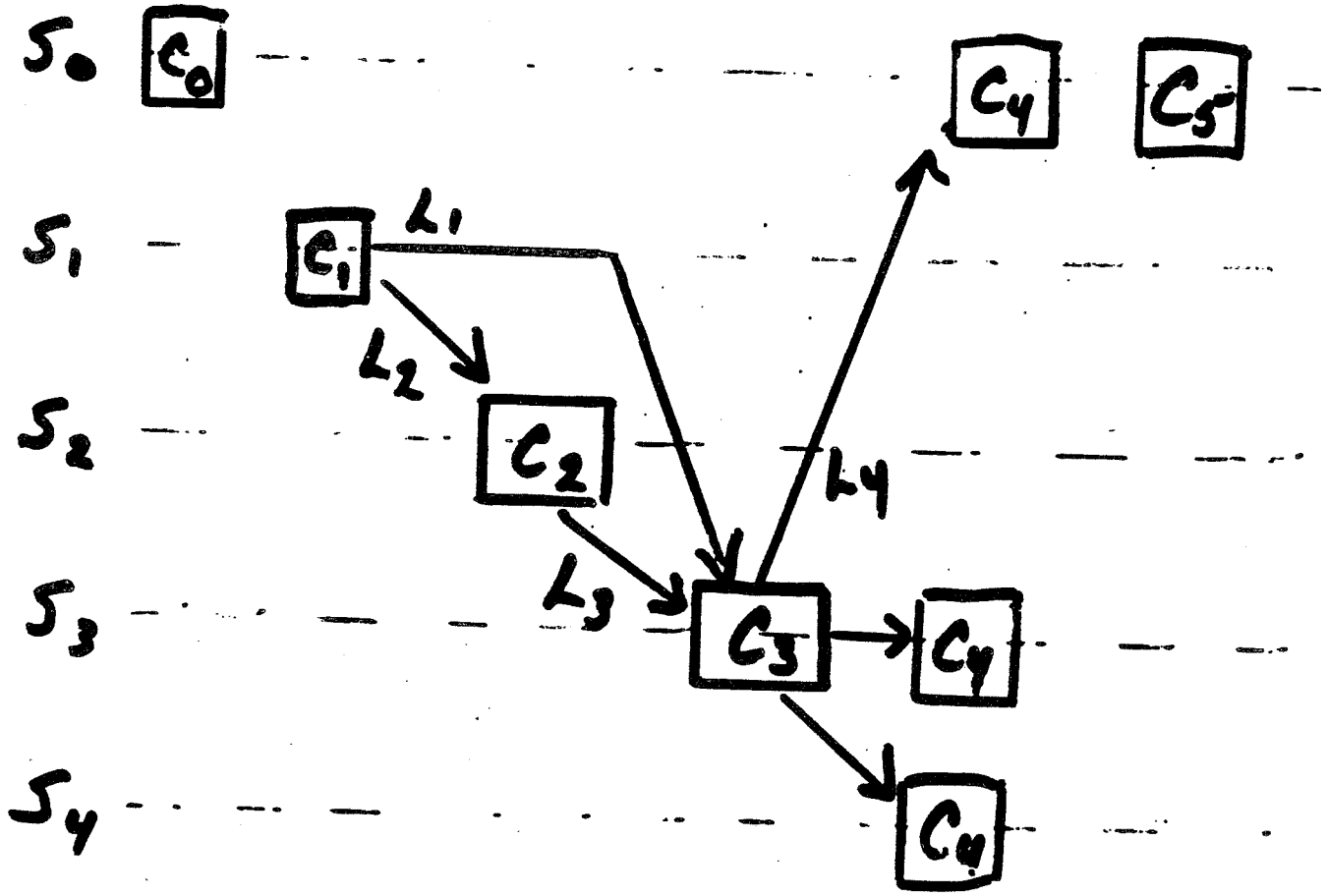
Site 0: z
Site 1: x
Site 2: y
Site 3: y,z
Site 4: x,z

Feasible locations of components

Components 0, 3, 5: Anywhere
Component 1: Site 1 or Site 4
Component 2: Site 2 or Site 3
Component 4: Sites 0, 3, AND 4
(Write ALL!)

For DYNAMIC DECISIONS, Directory must be fully replicated.

Execution Graph
(Mapping of Components to Sites during Execution)



- Optimizing the decisions.

IN SUMMARY:

Distributed, Passive Replicas

Replicas of Entire Transaction

Replica Components Activated Dynamically

Control, Execution History Migrates

Potential for Easy, Rapid Recovery

Scheduling of Imprecise Computations

Kwei–Jay Lin and Albert Yu

Department of Computer Science
University of Illinois at Urbana–Champaign
Urbana, Illinois 61801

1. Hard real–time systems
2. Imprecise computation model
3. Scheduling algorithms

Real–Time Systems

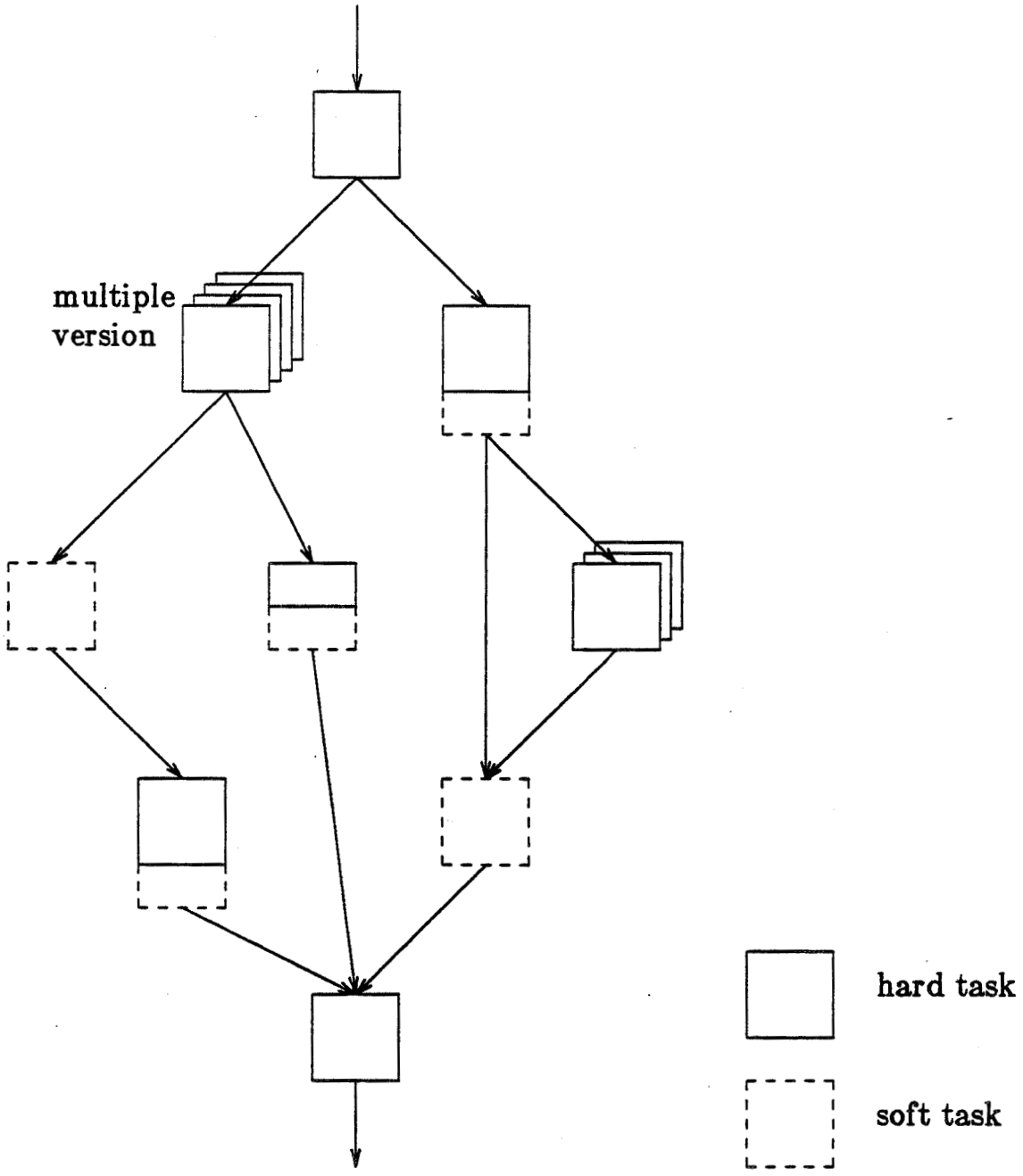
- Computations with timing constraints
- Hard real–time systems
  - Late results are of little or no use
- Predictable real–time systems
  - Produce acceptable results on time
  - Graceful performance degradation under fault–tolerant requirements
- Flexible real–time systems
  - Imprecise computation technique
  - Parallelizable tasks

Examples of Real–Time Systems

- Avionic coordination systems
- Patient monitoring systems
- Intelligent highway vehicle systems for national transportation
- Transmission of digitized picture or voice
- Farming equipments for fertilizing soils
- Nuclear power plants

Imprecise Computation Model

- Nodes: imprecise tasks with timing constraints
- Arcs: dependence and AND/OR relations



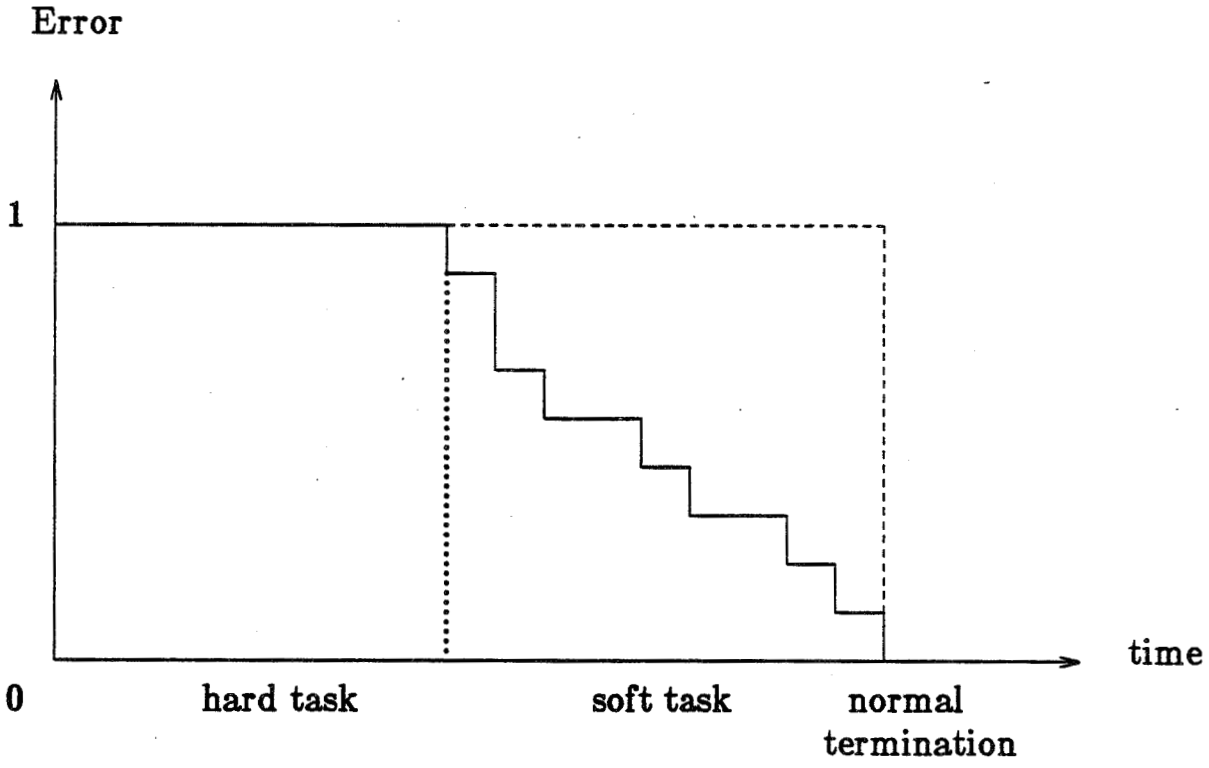
Scheduling Algorithms for Imprecise Computations

- Algorithms for scheduling aperiodic task with ready times, deadlines and precedence constraints on uniprocessors
  - to minimize the total error
  - to minimize the maximum error
  - to minimize the mean flow time
  - to minimize the number of discarded optional subtasks
- Algorithms for scheduling aperiodic independent tasks with ready times and deadlines
  - to minimize the total error on multiprogrammed multiprocessor systems
  - to minimize the total error when parallelized
- Algorithms for scheduling periodic task, that are
  - error–noncumulative, to minimize average error
  - error–cumulative, to meet all deadlines

- Task assignment algorithms for replicated imprecise tasks
- 2–level queuing polices for optimal tradeoff between average response time and result quality
- A monotone query processor for relational algebra queries

Monotone Imprecise Computation

- Ready time (r^j)
- Deadline (d^j)
- Complete processing time (τ^j)
- Processing time of hard task (h^j)
- Weight (w^j)
- Error (ϵ^j)



Available Monotone Algorithm and Methods

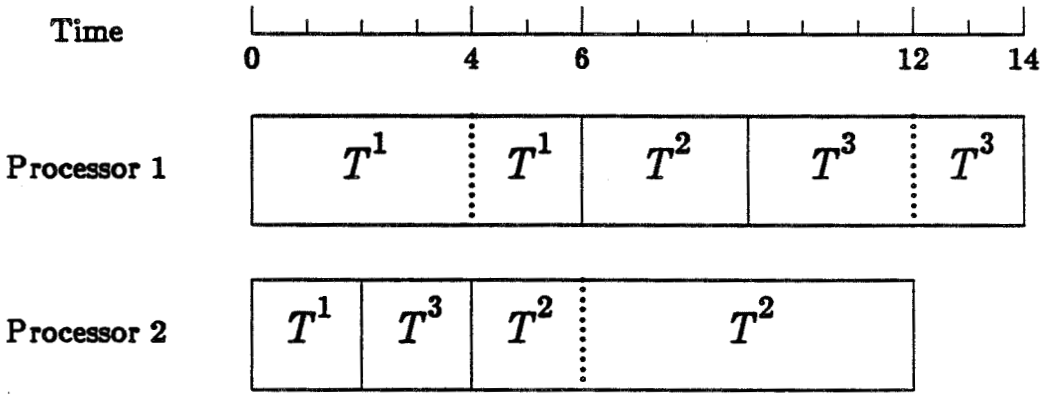
- Iterative algorithms
- Transmission of compressed digitized picture or voice
- Progressive buildup method for facsimile transmission
- Successive doubling method for computing FFT's
- Multiple phase algorithms
- Statistical methods, e.g., Monte Carlo Method
- Approximate relational algebra query processing

Parallelizable Tasks

- Multiprocessing overhead (o^j)
- Maximum degree of concurrency (K^j)
- An example task system

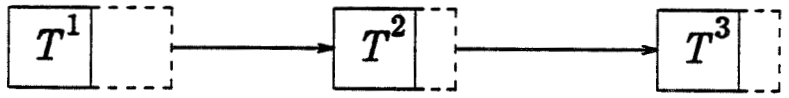
	r	d	τ	h	s	w	o	K
T^1	0	6	8	3	5	3	2	2
T^2	4	12	13	10	3	2	2	2
T^3	0	14	17	7	10	1	2	2

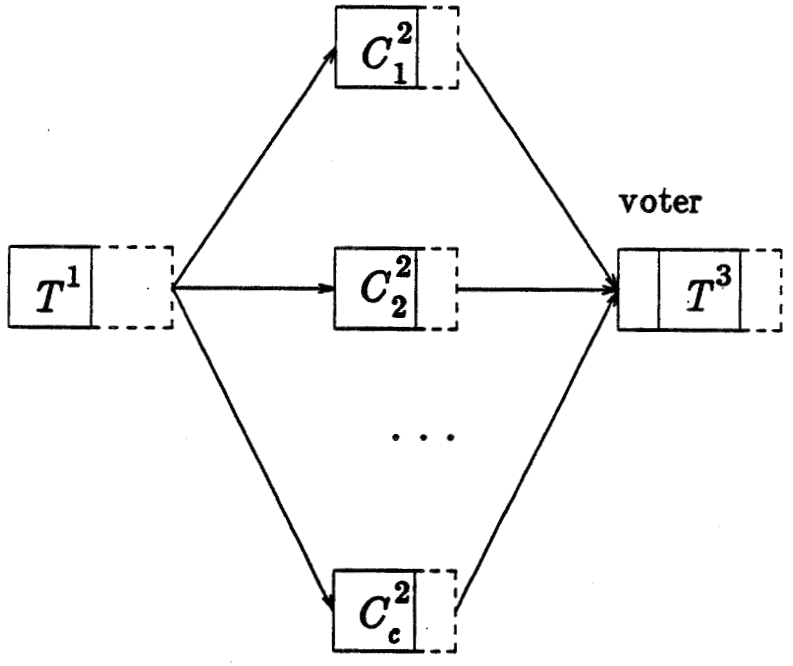
- Multiprocessor schedule



Fault Tolerance

- Two general approaches
  - Rollback and retry
  - Redundancy and masking
- Replicated Task Structure





Algorithms for Allocating Replicated Periodic Tasks

- Allocation algorithm
  - Seeks a good initial task assignment close to the optimal
- Modify algorithm
  - Finds a new task assignment with the least change from the old one, as PE's fail or recover

Summary

- Flexible real–time systems produce acceptable results on time
    - Imprecise computation
    - Parallelizable tasks
- Fault tolerance
    - Replication and masking
- PERTS

PERTS

A Prototyping Environment for Real-Time Systems

- contains real-time system building blocks, together with analysis and evaluation tools to support their use.
- can be integrated into an object-based software engineering environment and tailored for different application domains and operating systems

J. W. S. Liu, K. J. Lin and C. L. Liu

Real-Time Systems Laboratory
Department of Computer Science
University of Illinoi, Urbana, IL

- **Basic Research Accomplishments**

  —*Algorithms and Protocols*

  - Imprecise computation paradigm: computational algorithms and scheduling algorithms
  - New workload models and scheduling algorithms: AND/OR scheduling, distance-constrained scheduling, end-to-end scheduling and monitor-based scheduling
  - Resource management policies: for dynamic-priority scheduling and serializable accesses

  —*Methods, Tools and Simulators*

  - Worst-case performance analysis: a systematic method, worst-case performance bounds for common task systems
  - Performance profiling: workload models and simulators
  - Timing analysis and measurement tools

  —*Programming Languages and Environments*

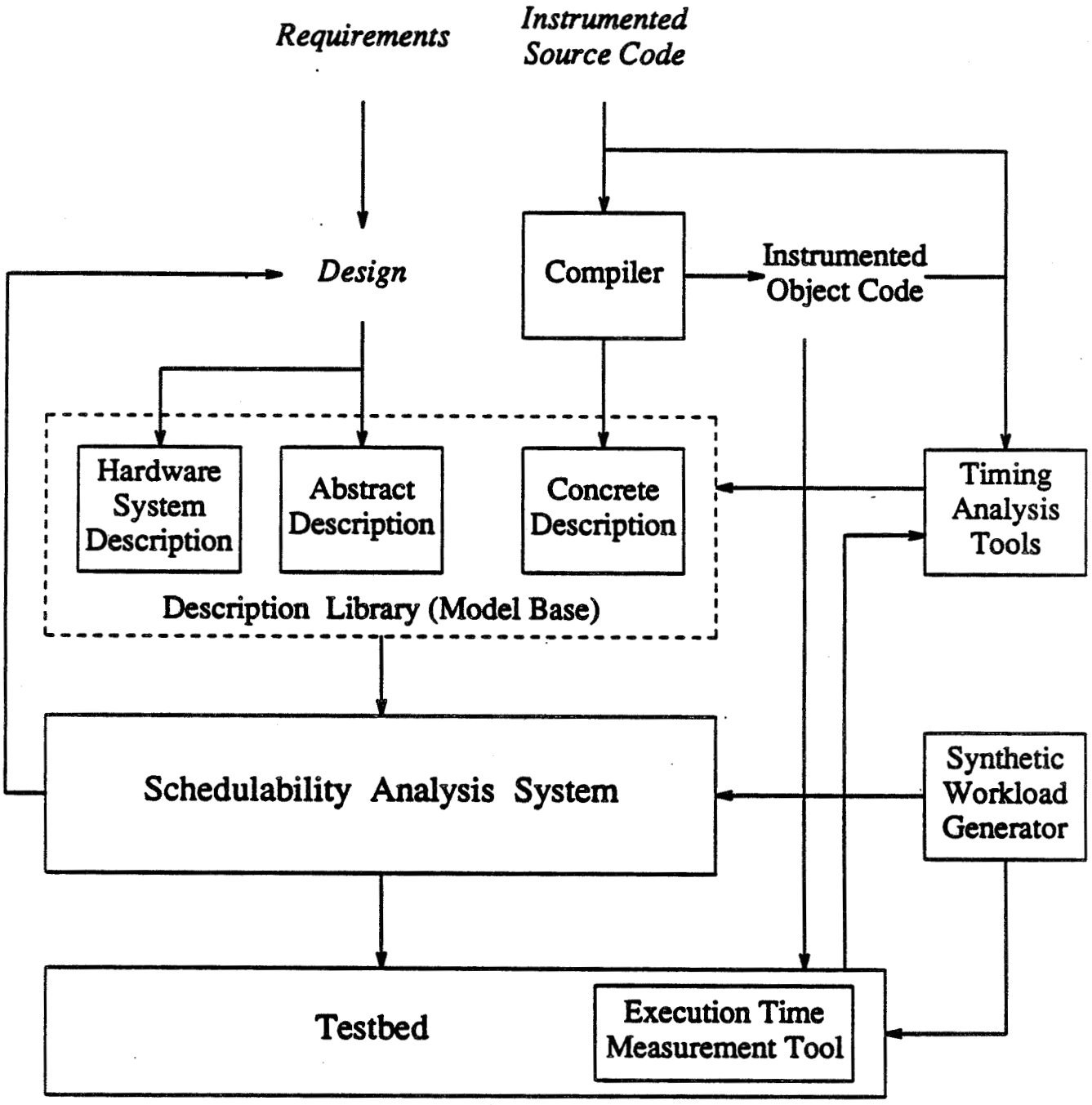
  - Flex, an environment for writing flexible, real-time programs
  - Mentat, an environment for data-driven execution of real-time programs

- **Goals of the Current Work**

  —*Integration of the results*: providing complete solutions

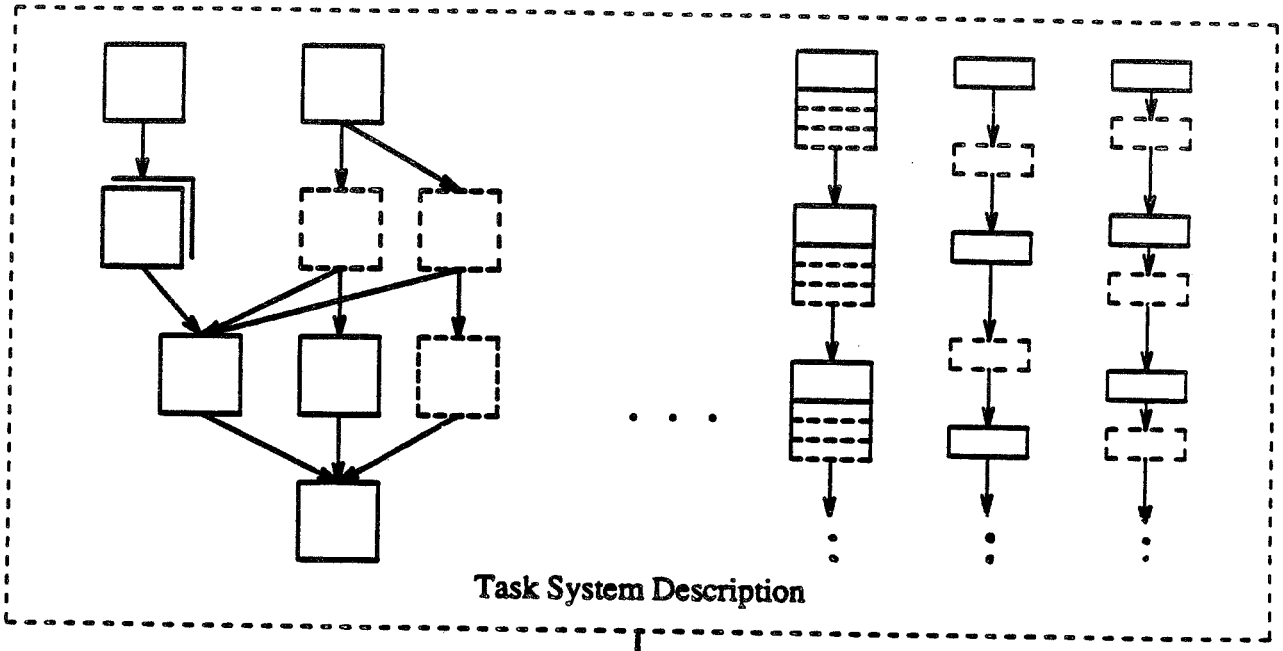
  —*Technology transfer*: putting the research results in usable forms

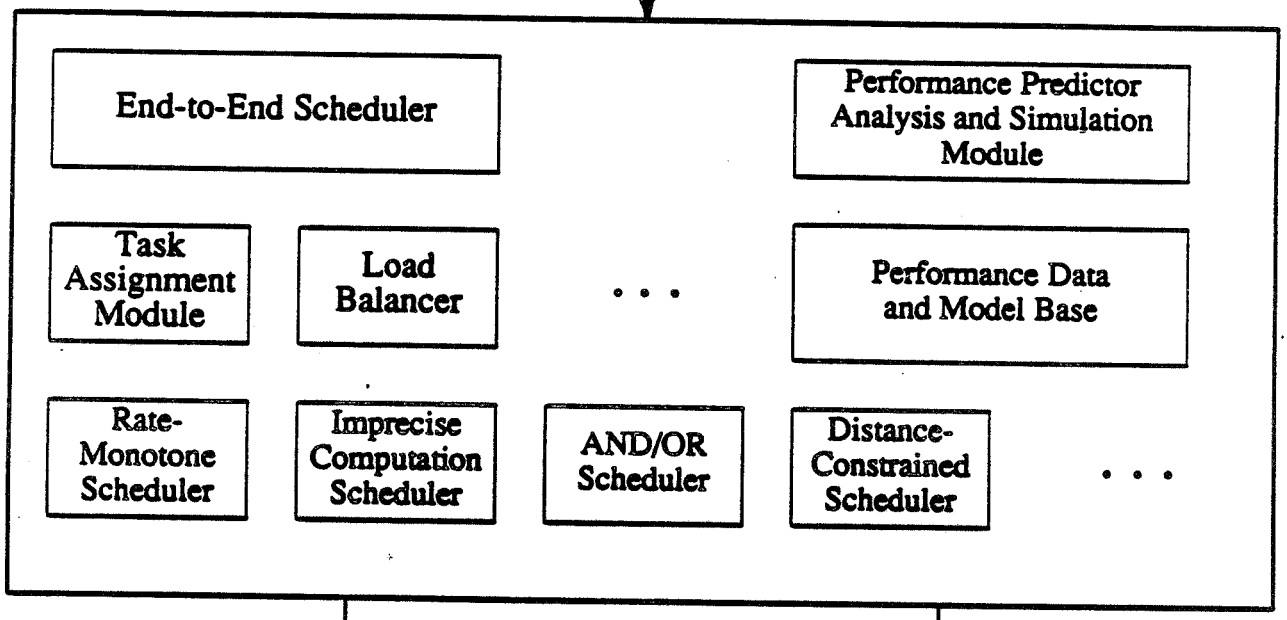
PERTS


Requirements
Instrumented Source Code
Design
Compiler
Instrumented Object Code
Hardware System Description
Abstract Description
Concrete Description
Timing Analysis Tools
Description Library (Model Base)
Schedulability Analysis System
Synthetic Workload Generator
Testbed
Execution Time Measurement Tool

- As an *interactive design tool*
  - — The user provides
    - task system description (an annotated data-flow graph of the task system),
    - hardware system description, and
    - information on additional external constraints.
  - — Outputs produced by PERTS include
    - processor and resource requirements,
    - sample task partitions and allocations,
    - sample schedules and memory layouts,
    - performance predictions, and
    - suggested design changes and tests.
- As a *development and evaluation tool*
  - — The user provides
    - annotated source code or object interface definitions, and
    - system description.
  - — PERTS can provide
    - a simulated (or emulated) target environment, and
    - performance profile.

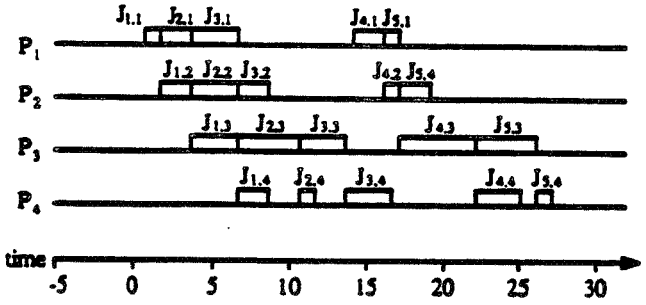
Schedulability Analysis System





Sample Schedule

Scheduling Directives to Testbed



Building PERTS Incrementally

- Implementation of the user interface
  - — Interfaces and tools for the input and visualization of
    - target task systems
    - system resources
    - configuration constraints
  - — PERTS language and compiler

- Implementation and porting of basic building blocks:
  - — algorithms for hierarchical decomposition, task assignment, and end-to-end scheduling
  - — suites of algorithms that support different scheduling and resource management paradigms

- Implementation and porting of performance profiling tools:
  - — timing analysis and measurement tools
  - — generic workload models
  - — simulators and emulators

- Building of a database of workload models, system models, representative target systems, schedulers, protocols, etc.

Examples of PERTS Components

— building blocks of imprecise computation systems

— rate-monotone and modified-rate-monotone schedulers

— end-to-end scheduler

— AND/OR scheduler

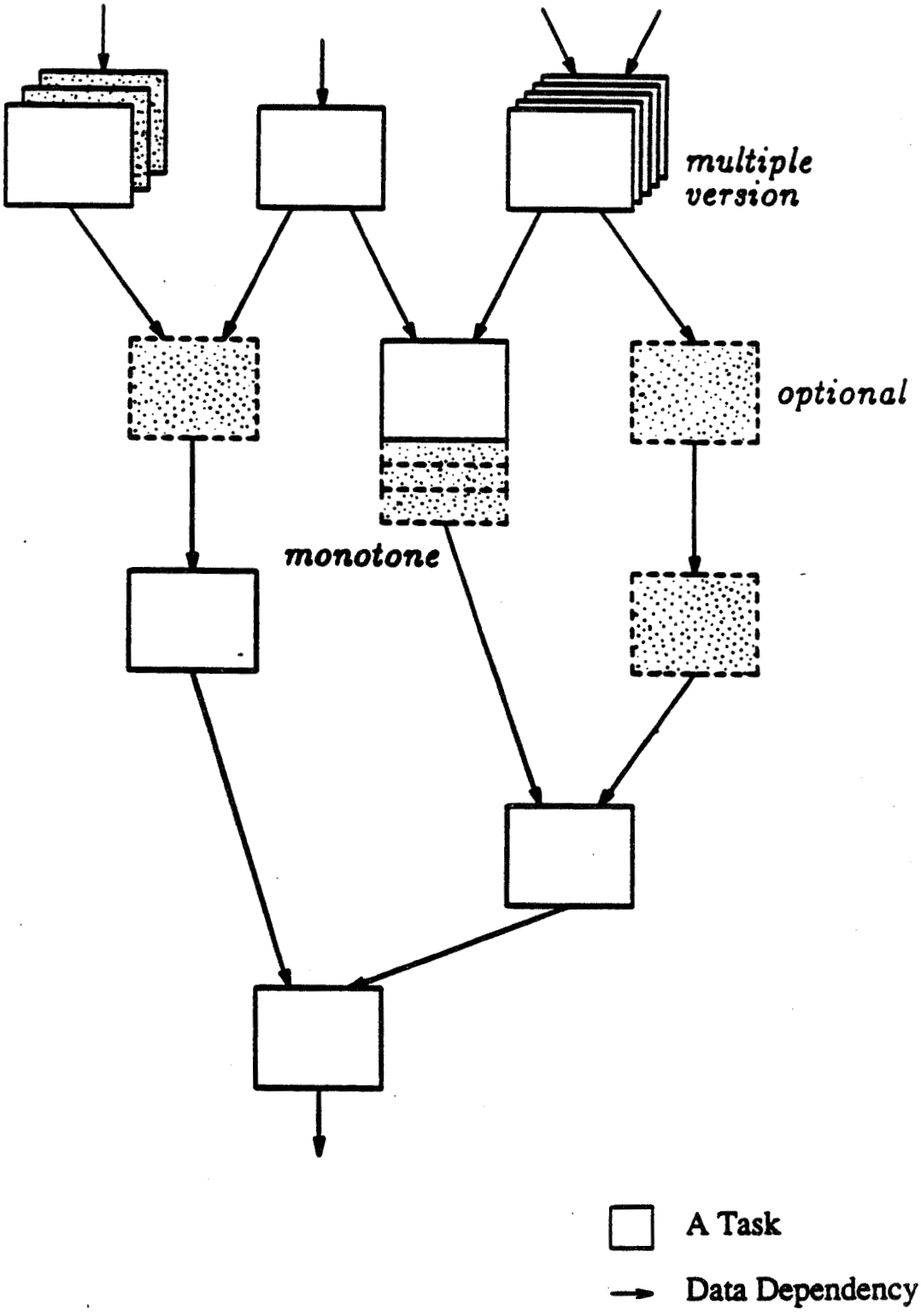
— distance-constrained scheduler

— powerful performance predictor

— static and dynamic timing analysis tools

— versatile simulation (and emulation) environment

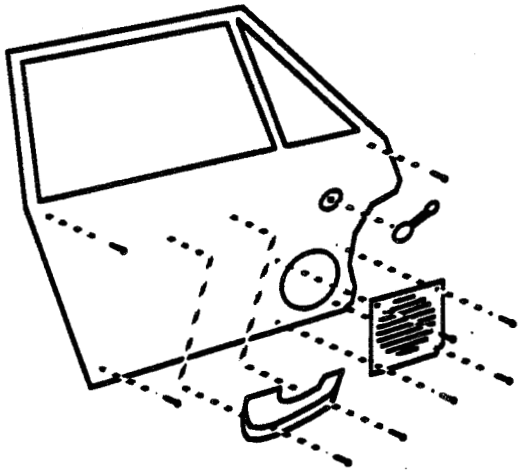
The Imprecise Computation Model



Available Building Blocks of Imprecise Computation Systems

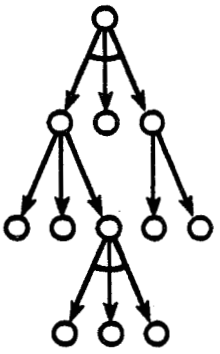
- Algorithms for scheduling aperiodic tasks with ready times, deadlines and precedence constraints on uniprocessors
  - — to minimize the total error
  - — to minimize the maximum error
  - — to minimize the mean flow time
  - — to minimize the number of discarded optional subtasks
- Algorithms for scheduling aperiodic independent tasks with ready times and deadlines
  - — to minimize the total error on multiprogrammed multiprocessor systems
  - — to minimize the total error when parallelized
- Algorithms for scheduling periodic tasks, that are
  - — error-noncumulative, to minimize average error
  - — error-cumulative, to meet all deadlines
- Task assignment algorithms for replicated imprecise tasks
- 2-level queuing policies for optimal tradeoff between average response time and result quality
- A monotone query processor for relational algebra queries

AND/OR Scheduling Applications



Manufacturing Assembly

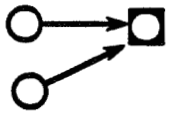
Algebraic Systems
(w/Distributive, Associative,
Commutative laws)



AI/Heuristic Search
(directed Alpha-Beta search)

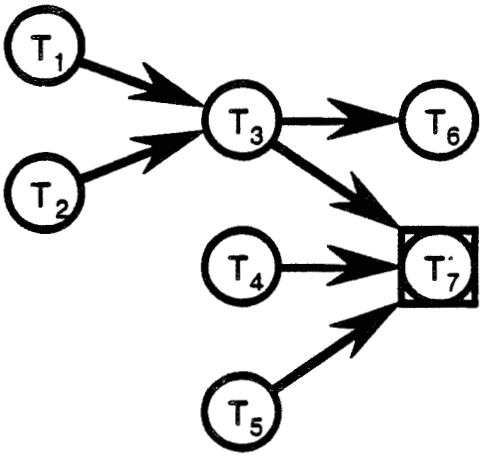
Mandatory	Optional

Imprecise Computation
- Error Cumulative
- Error Non-Cumulative

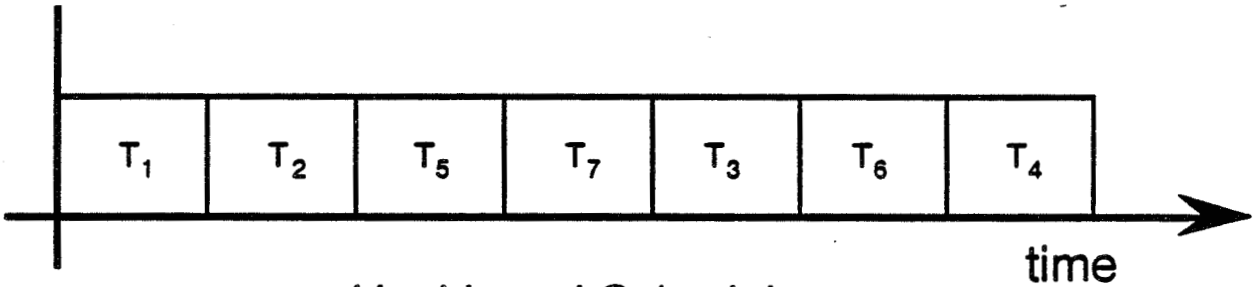


Limited Resource
Scheduling

AND/OR Sample Schedule

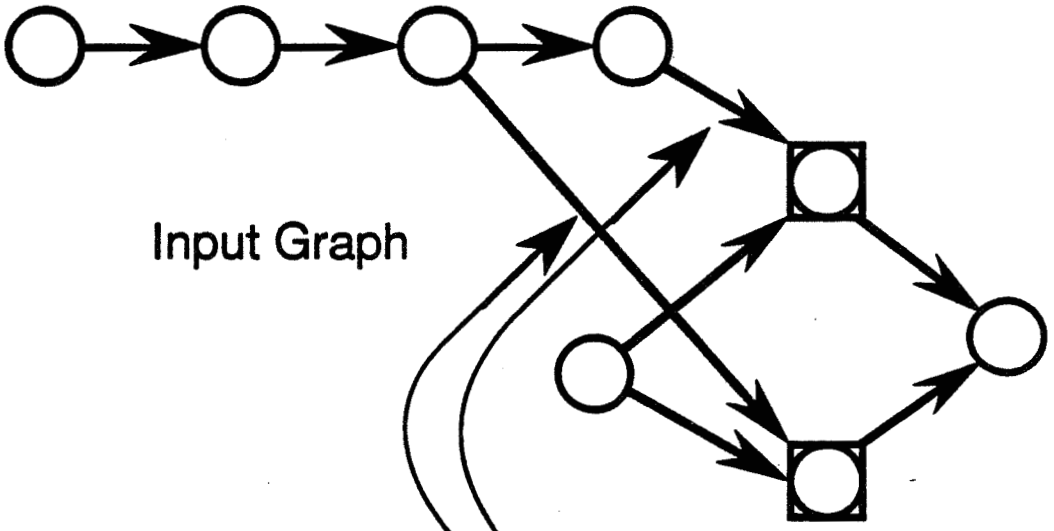


Task System

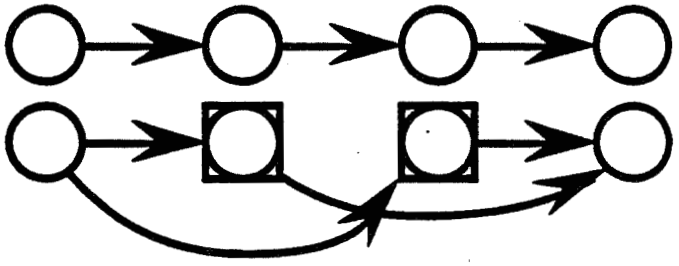


Unskipped Schedule

AND/OR/Unskipped Scheduling Algorithm

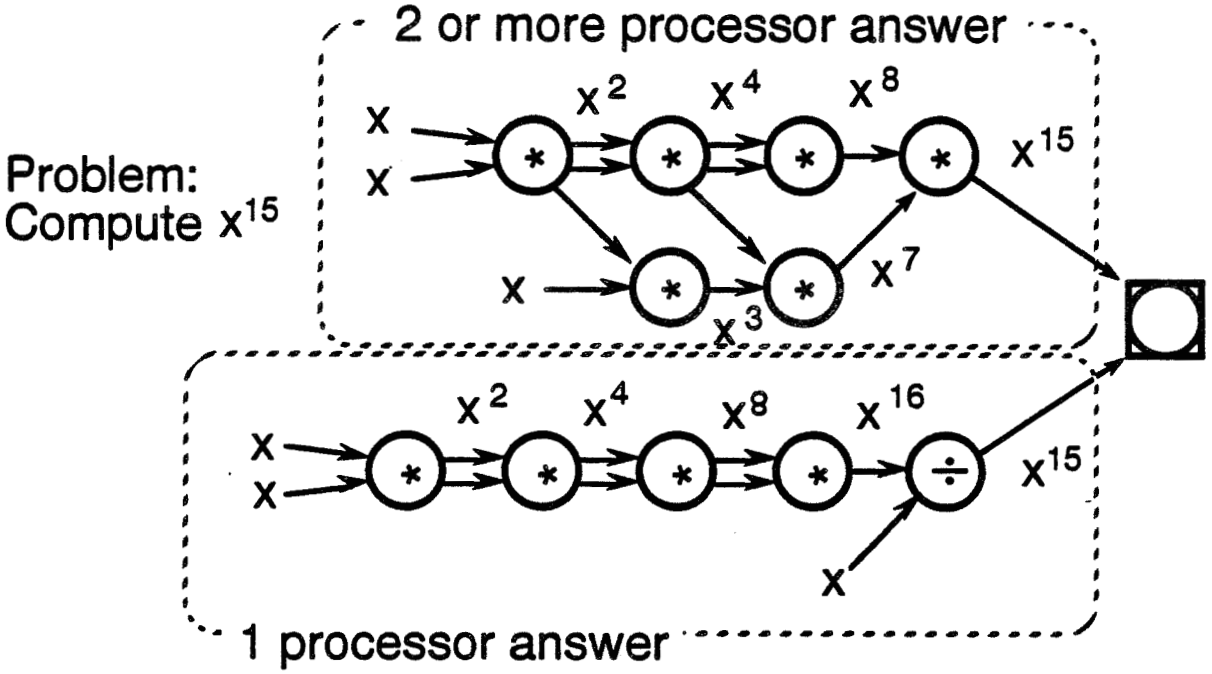


1. For each OR Task:
   Break the arc(s) that begin
   the longer path(s)



2. Schedule the resultant task system
   on m processors (m = 2 shown)

AND/OR/Skipped Scheduling Algorithm

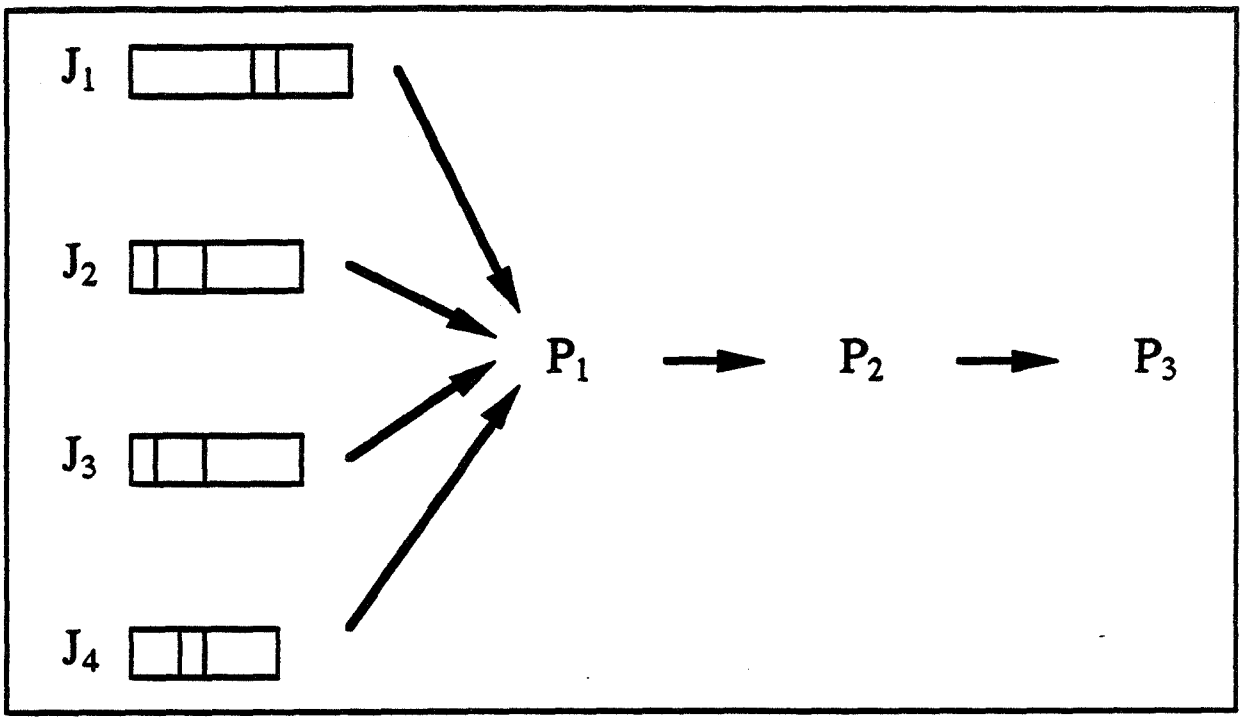


1. Enumerate all longest paths from root task to leaf
2. $L^*(G)$ is the length of the longest path in G, the task graph.

   $E^*(G)$ is defined as $(\Sigma p_i - L^*(G))/m$

   Choose the subgraph that minimizes $E^*(G) + L^*(G)$.

Flow Shops

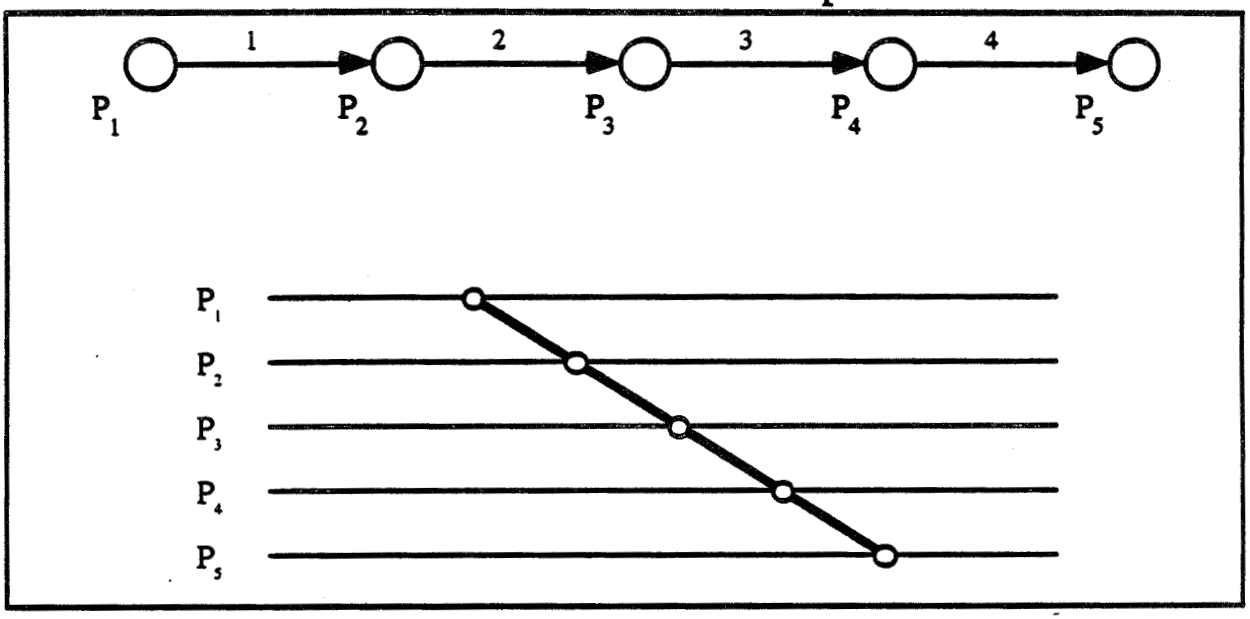


Further Applications:

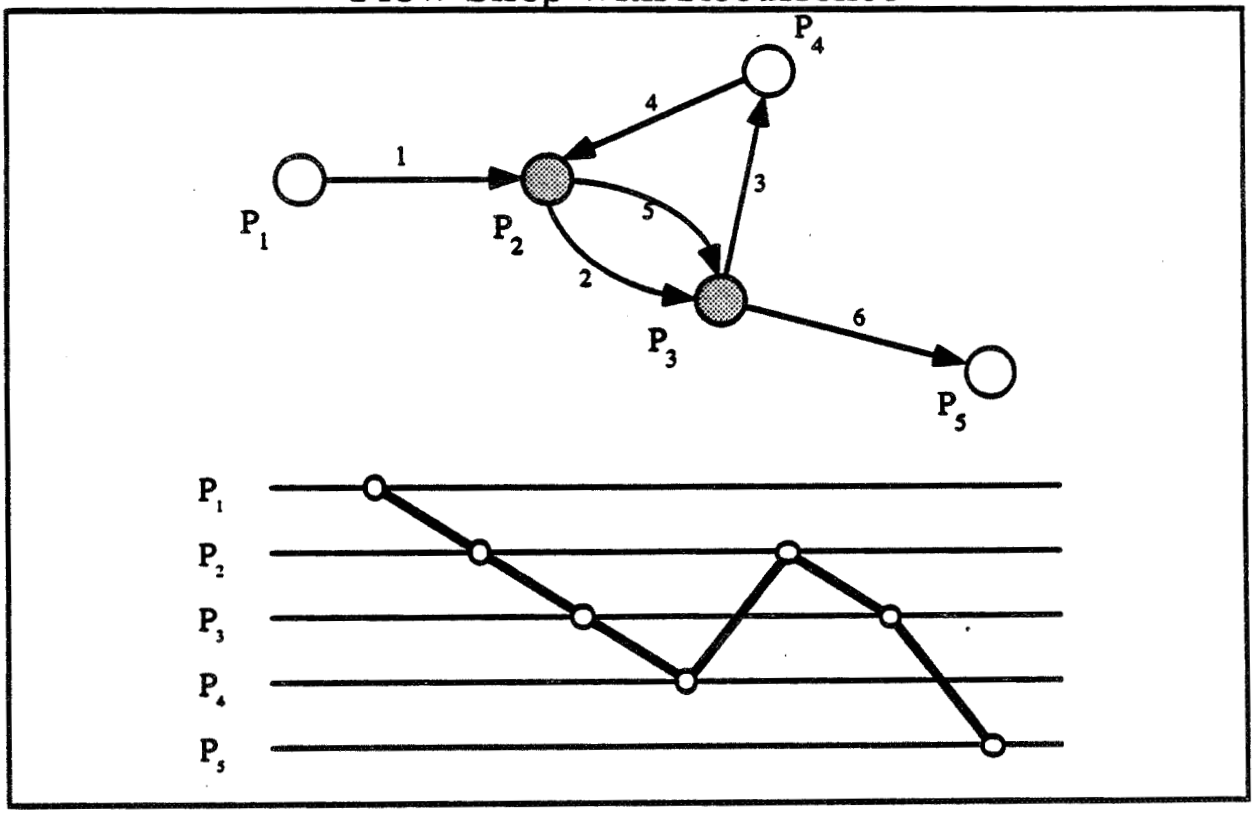
- Functionally dedicated processors
- Data transmission systems
- Instruction scheduling
- Logic circuit design

Flow Shop with Recurrence

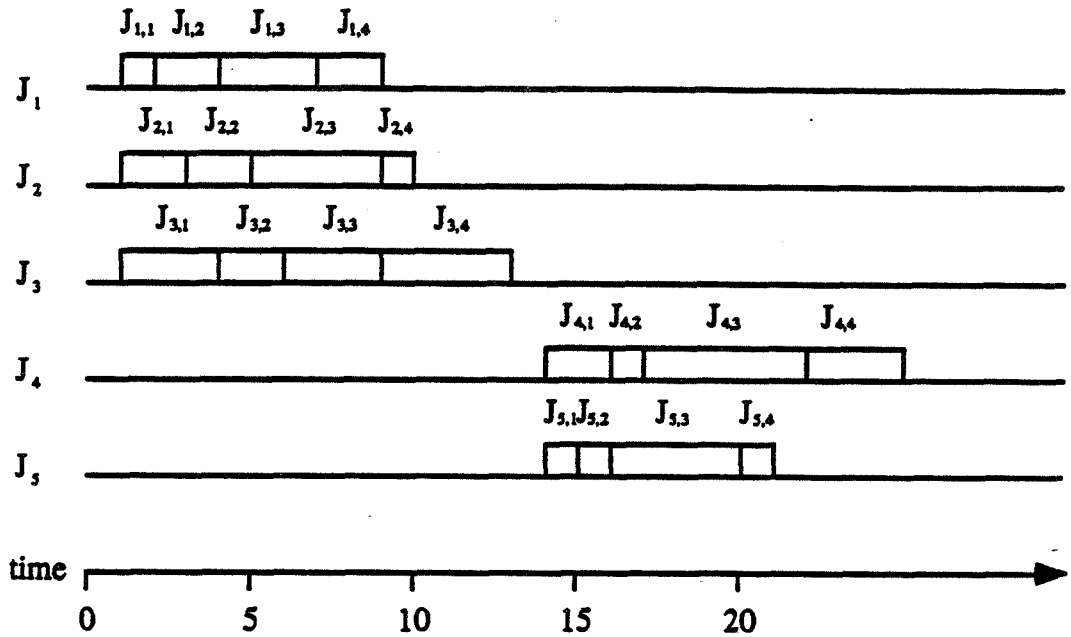
Traditional Flow Shop

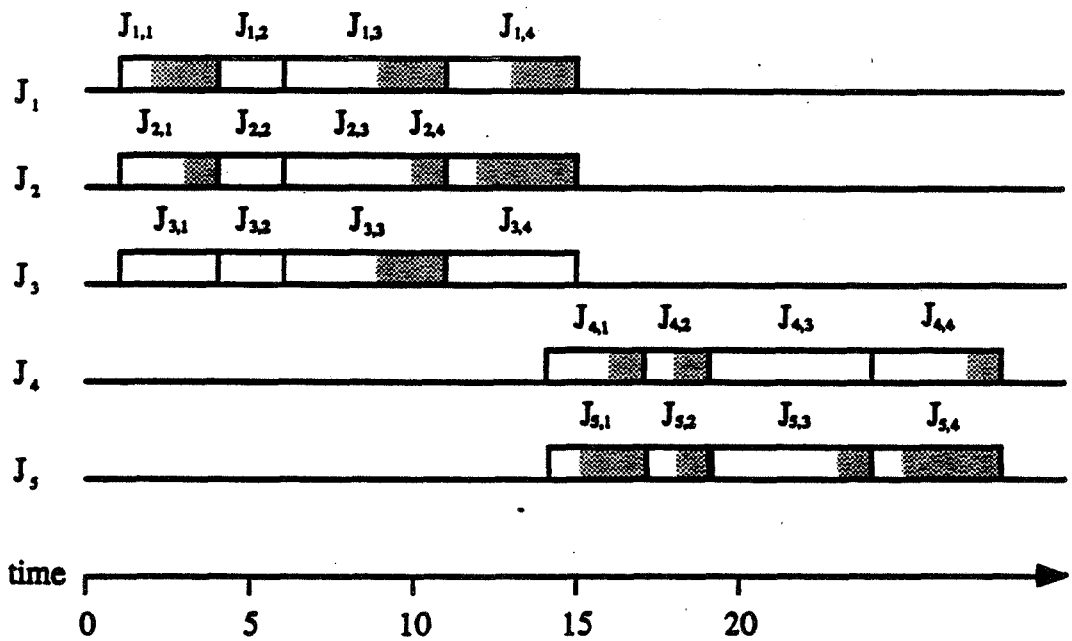


Flow Shop with Recurrence

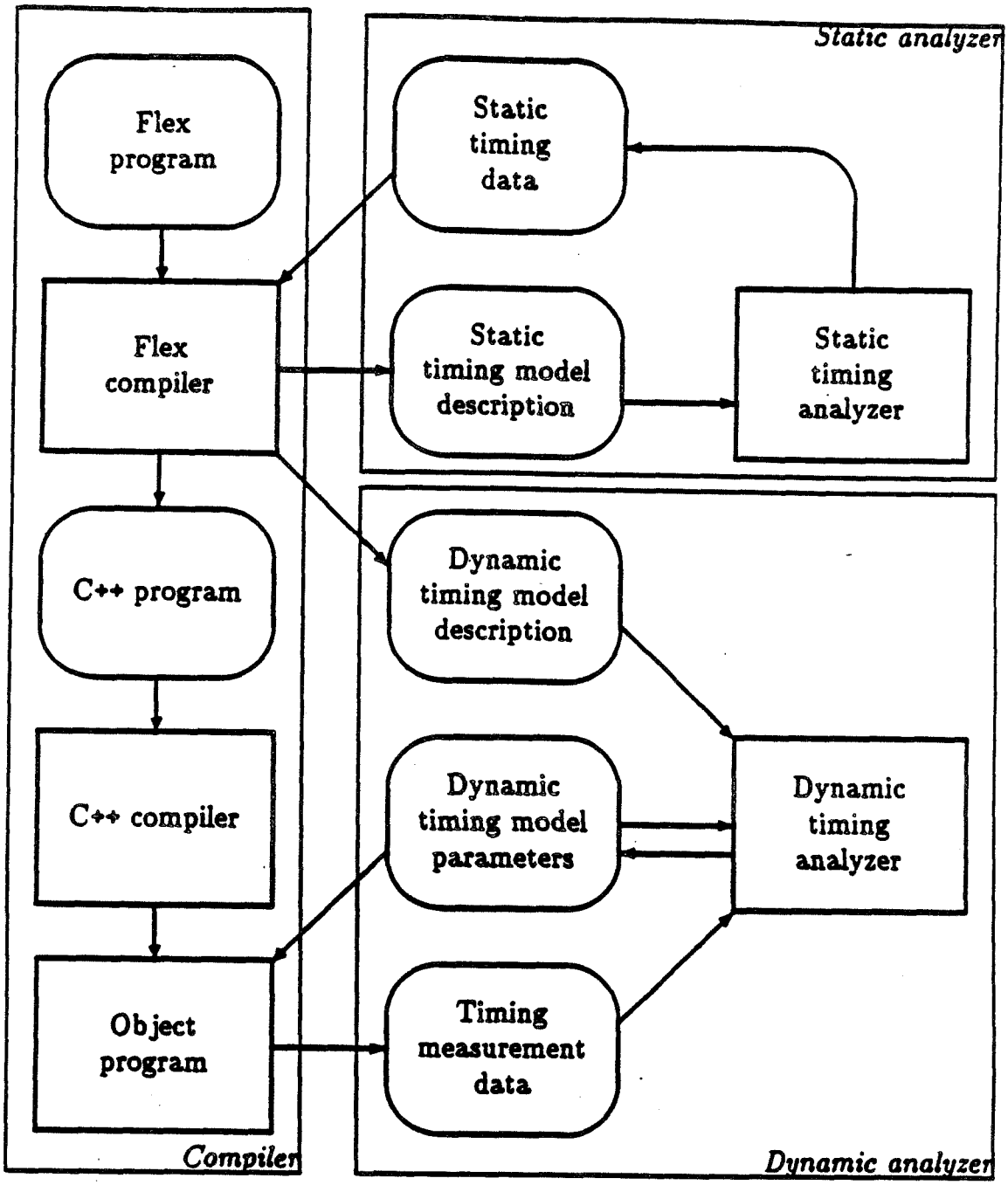


Scheduling Arbitrary Flow Shops to meet Deadlines

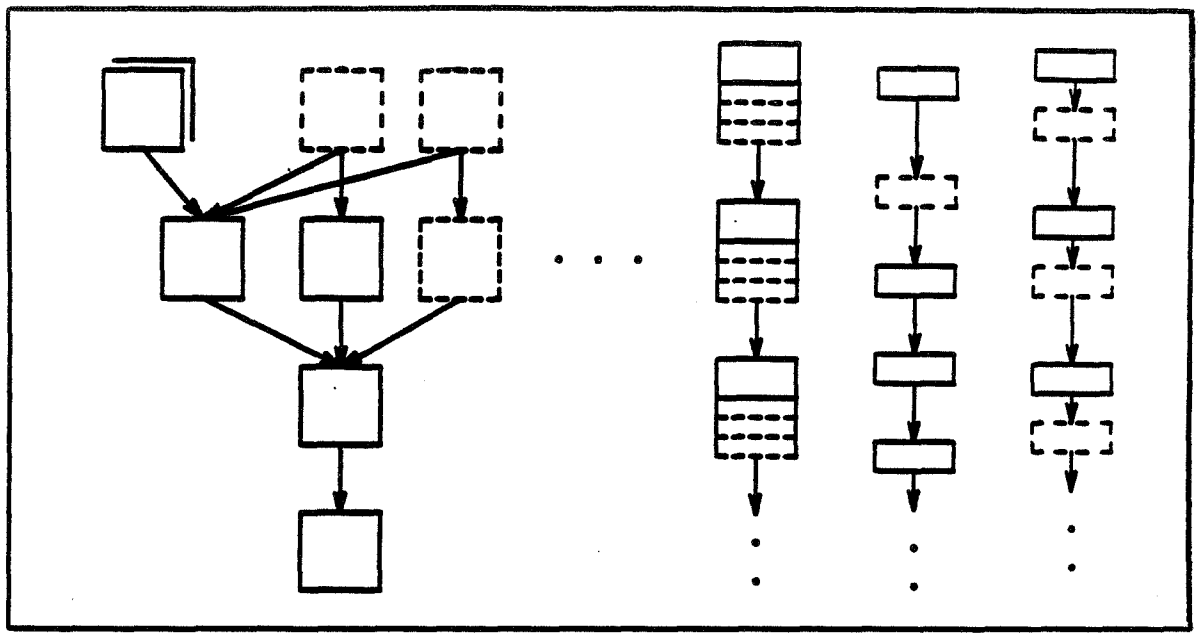




Timing Analysis Tools



PERTS Execution Environment Architecture



Virtual Machine				
Token Matcher (Uninstantiated)	Token Matcher (instantiated)	Runtime Dataflow Detection	Computation Units	Scheduler

Machine Specific Components		
Intra-processor communication	Inter-processor communication	Host OS Interface

- Object-oriented, with persistent objects, and data-driven
- Timing mechanism and imprecise mechanism

PERTS Extensions to C++

— the keyword **perts**

— **creat()/destroy()** statements

— return to future (**rtf()**)

— **select/accept** statements

— timing blocks and timing constraints

PERTS Timing Blocks

- attributes of timing constraints:
  - — soft vs hard
  - — start time and deadline
  - — duration
  - — period

- restrictions for actors with
  - — hard timing constraints: static graph with bounded execution time and resource requirements
  - — soft timing constraints: no restrictions

- an example

```
soft (duration > 1.0) ~> rtf()
    begin
        while(true)
        some statements
        end;
    end;
```

```
(period = 1.0) ~> end_iteration
    begin
        (duration > 0.8) ~> exit_block
            begin
                (duration > 0.5) ~> exit_block
                    begin
                        select {
                            :accept receiver1();
                            r1 receiver1();
                            break;
                        };
                    end;
                (duration > 0.5) ~> exit_block
                    begin
                        select {
                            :accept receiver2();
                            r2 receiver2();
                            break;
                        };
                    end;
            end;
            select {
                (duration > 1.0-get_time()+start) ~> exit_block
                    begin
                        :accept {analyze(); analyze(r1,r2),
                        quick_analyze(); quick_analyze(r1,r2);} break;
                    end;
            }
    end;
```

GST: Grain-size Tuning for Efficient Execution

Professor Andrew A. Chien

ICLASS Review, Fall 1991

The Problem

Grain-size Transformation

Current Status, Directions

The Problem

Achieving efficient execution of fine-grained, concurrent object-oriented programs.

Motivation

- High Performance
  (sensor data processing, decision support, control)
- Modularity, reusability
  (Object-orientation, encapsulation)
- Portability
  (variety of architectures, parallel, distributed, and sequential execution)

Why is efficient execution difficult?

- fine-grain concurrency
- parallel or distributed machine (access or message latency)

Factors:

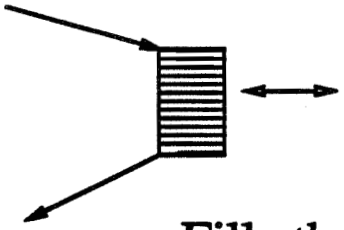
Machine Architecture

Machine configuration

Key: Adjust grain-size of program to suit the machine.

What is a computation grain?

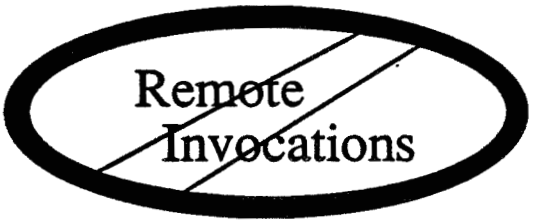
A computation "grain"



R/W local memory

Fills the pipeline without disruption.
Unit of scheduling.

Characteristics of a grain:

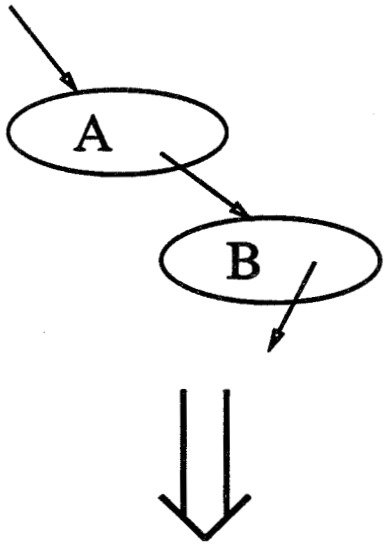


-> match grain-size to that most efficient for machine & problem

-> optimize control structure, data locality simultaneously

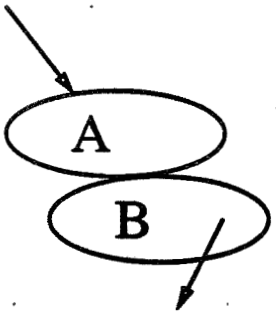
Idea: Grain-size Transformation

- code specialization/optimization
- contraints on object placement



Invocation

1) message to A
2) computation at A
3) message to B
4) computation at B
5) message back to A
6) computation at A
7) message back to caller



Invocation

1) message to A&B
2) computation on A&B
3) message back to caller

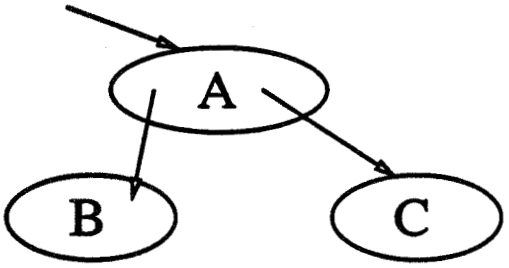
A,B Merged

A-B coupling overhead elim.

(larger grains, locality)

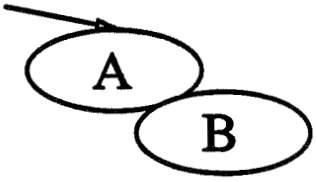
Challenges to Grain-size Transformation

- static analysis in modular, object-oriented programming system



"Alias Graphs"
"Locality Sets"

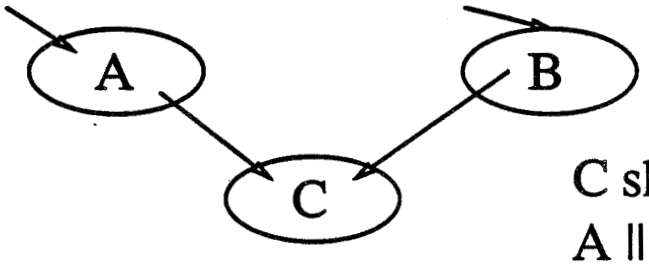
- implementation of transformations in dynamic allocation environment



"Co-allocation"
Code specialization

Challenges to Grain-size Transformation (continued)

- preserving concurrency / sharing semantics of program



C shared
A || C concurrency

- static transformation

- dynamic transformation
  (adaptation to actual constraints)

Benefits of GST approach:

Portability (efficient on # of platforms)

Modularity

High Performance

+ complex data structures allowed...

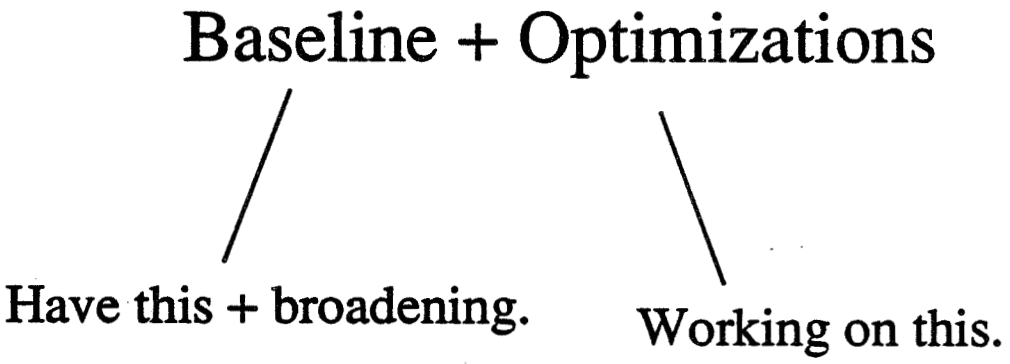
Current Status

Working Language Implementation

Coded application kernels (+ more)

Measurements of static/dynamic invocation patterns.

Optimization language and transformations



Performance Data Visualization and Sonification

Daniel A. Reed
reed@oboe.cs.uiuc.edu

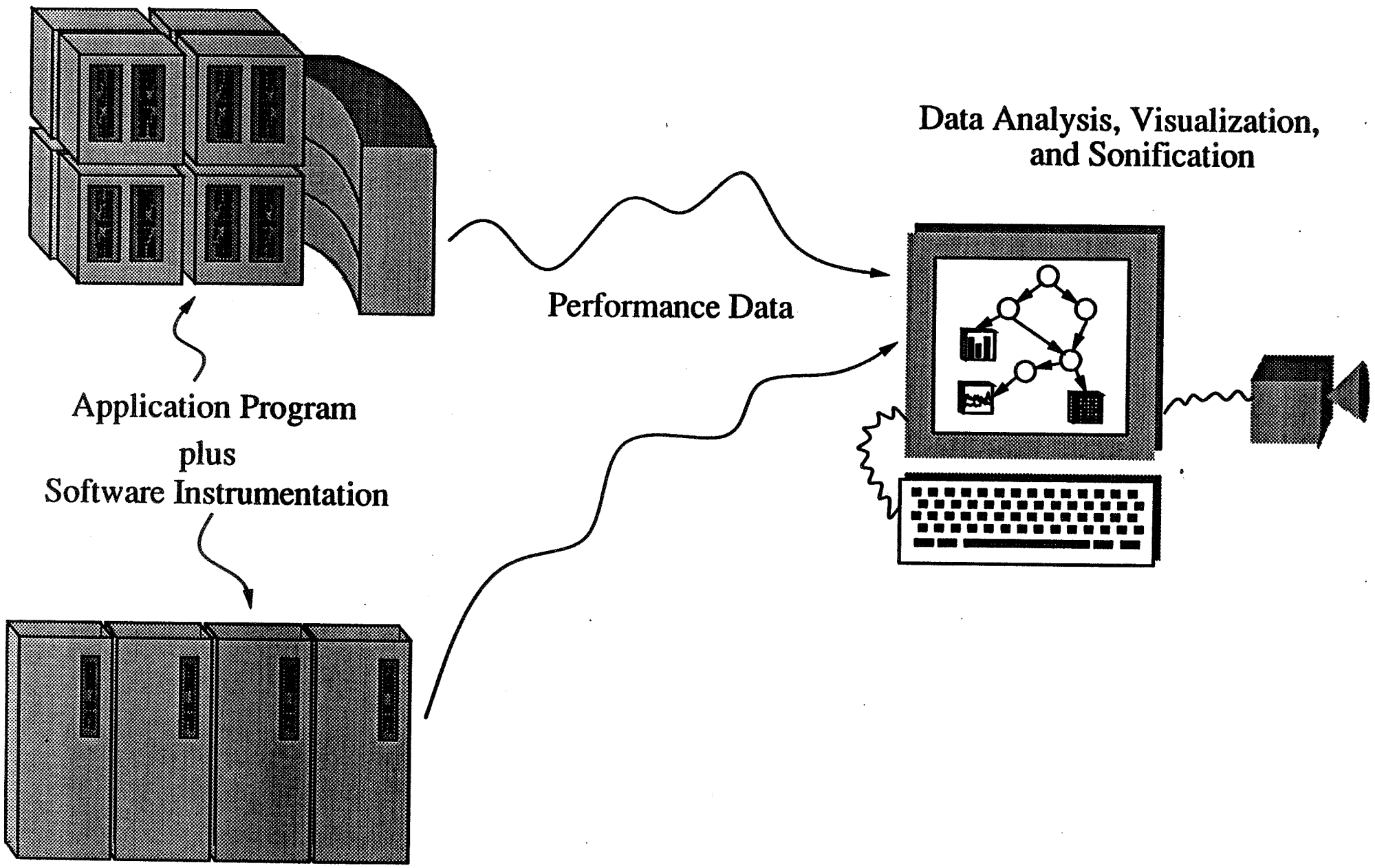
Department of Computer Science
University of Illinois
1304 West Springfield Avenue
Urbana, Illinois 61801

Overview

The goal of the *Tapestry* project is to devise system software that permits parallel applications to exploit the benefits of a heterogeneous collection of shared memory and message-based parallel processors.

Research Topics:

- Performance measurement and evaluation
- Performance visualization
- Resource allocation policies
- Structure of customizable operating systems (mechanisms)
  - uniform application interface
  - range of computer architectures
  - tailored policies and mechanisms


Data Analysis, Visualization,
and Sonification
Performance Data
Application Program
plus
Software Instrumentation

A Simple Gedanken Experiment

- Select your ten favorite application programs
- Measure the execution time of each on your favorite workstation
- Rank the applications based on measured execution time
- Repeat the process for another workstation

$\Rightarrow$ *The ranking is nearly invariant*

- Now repeat the process, but use parallel systems

$\Rightarrow$ *This is little correlation between rankings*

- We must understand the interactions of
  - application software
  - system software
  - architecture

Performance Observability

Performance Observability is the ability to

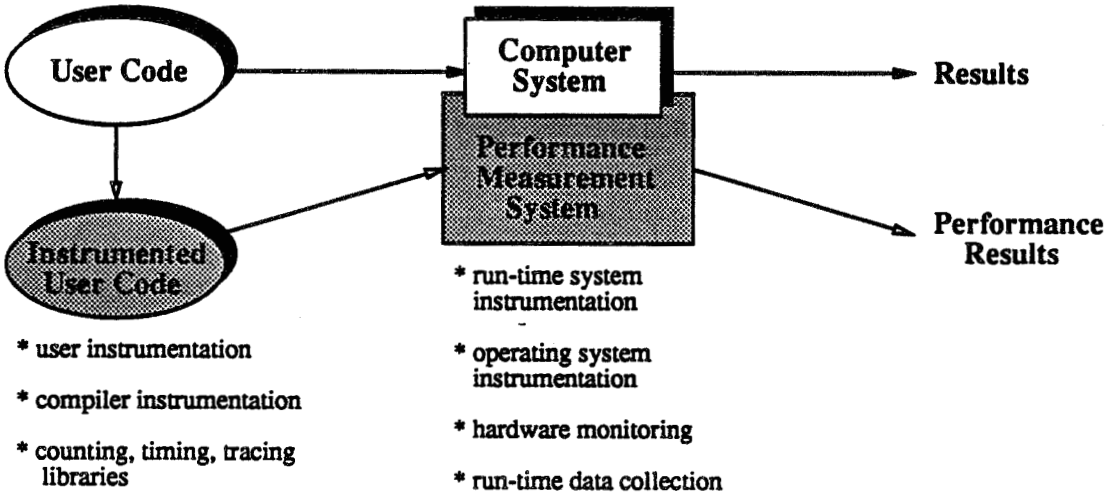
- capture
- analyze
- present

(collectively *observe*) information about computer performance

Tools for performance observability must balance the

- need for performance data
- cost of obtaining performance data

Performance Instrumented Computer



- Instrumentation and data collection are fundamental
- Must be included early in the computer system design
- Degree and type of instrumentation depend on intended purpose
- Economic considerations cannot be overlooked
- Few extant performance measurement systems on commercial machines

Performance Scenarios

- Speedup
  - coarse level
  - no insight
- Mean/distribution of parallelism
  - *when* — yes
  - *where* — no
- Time profiles
  - *when* — no
  - *where* — yes
- Timestamped event traces
  - *when* — yes
  - *where* — yes

$\Uparrow$ Accuracy $\Downarrow$ Complexity

Tracing versus Statistical Summaries

- Tracing is to software as a logic analyzer is to hardware
- Tracing captures dynamics
- Statistics can be calculated from traces
- Traces reveal periodic behavior
- Traces can be correlated across multiple levels
  - hardware
  - system software
  - application

Pablo Performance Environment — Motivations

- Parallel systems exacerbate performance analysis problems
  - additional software complexity
  - more performance variables

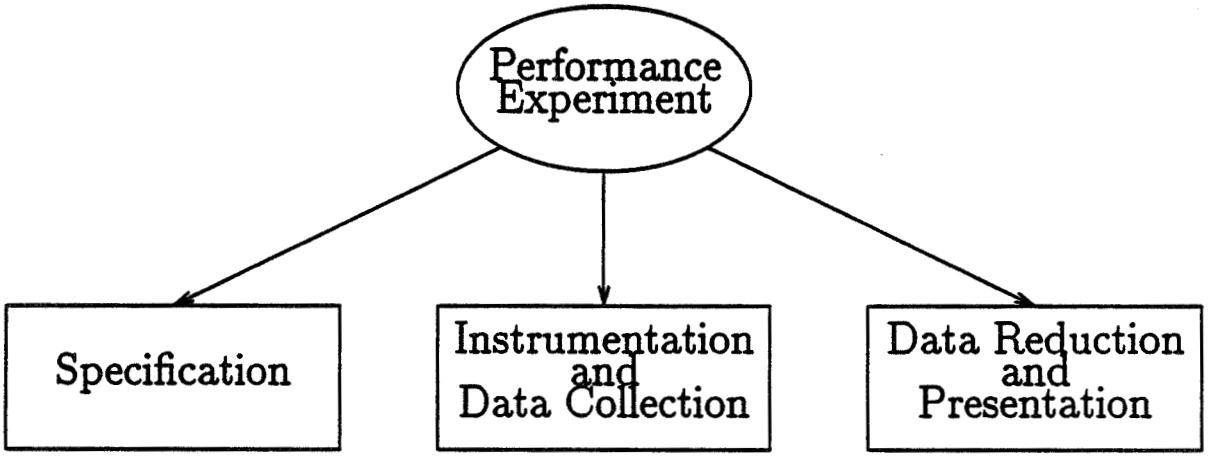
- Antithetic
  - data volume
  - data accuracy

⇒ Integrated performance environments
  - instrumentation specification
  - software data collection
  - analysis
  - visualization
  - sonification
    1. voice
    2. alarms
    3. dynamic sounds

Performance Visualization Environment Design Goals

- The environment should support **multiple analysis levels**, including hardware, system software, and application.
- It should be possible to **dynamically configure** data analysis and visualization components, allowing the performance analyst to change data perspectives during execution.
- The individual analysis and visualization components should be **easy to build** for many different application types and system software environments.
- An **extension interface** should permit addition of new components or modification of existing components without intimate knowledge of environment infrastructure.
- The environment should be **fast**, preferably fast enough to process real-time data.
- Finally, the environment should be **portable** to different systems. Although the mechanisms for performance data capture are inherently system dependent, performance data reduction (e.g., computation of sliding window averages) and visualization are largely system independent

Performance Analysis Phases



Tapestry

Pablo Performance Environment Components

- Portable software performance instrumentation
  - graphical event specification
  - source code instrumentation
  - portable data capture

- Performance data analysis and presentation
  - graphical programming
  - data transformation modules
  - self-documenting data format
  - dynamic graphics
  - sonification

Pablo Analysis Environment Design Motivations

- Performance data analysis software should
  - be ***extensible*** to new parallel systems
  - allow user ***configuration*** to support new types of data analyses
  - ***hide the details*** of data formats, storage, and manipulation

- Software embedded knowledge of parallel machine semantics precludes portability

- Fortunately, most performance data analyses are simple data transformations that do not require knowledge of the data source
  - sliding window averages
  - histograms
  - counts/sums

- Pablo performance environment will support
  - parallel, machine independent data analysis
  - graphical interconnection of data transformation modules
  - interactive selection of trace data records for analysis
  - dynamics graphics and sound for presentation of analyzed data

- A flexible data meta-format is the key to software portability

Standard Data Formats — Issues

- Compactness
  - traces can be large (megabytes or gigabytes)
  - data compression should not be precluded
- Portability
  - byte order
  - machine types and precision
  - physical media
- Binary vs. ASCII (ISO)
  - compactness
  - readability
  - processing speed
- Data semantics
  - implications for standard tools
  - portability
  - cross-architecture machine comparisons

Trace Data Diversity

- Hardware
  - TLB
  - Cache
  - Network
  - Memory
  - Functional Units
- Software
  - Opcodes
  - memory references
  - operating system activity
  - application control flow
  - application data

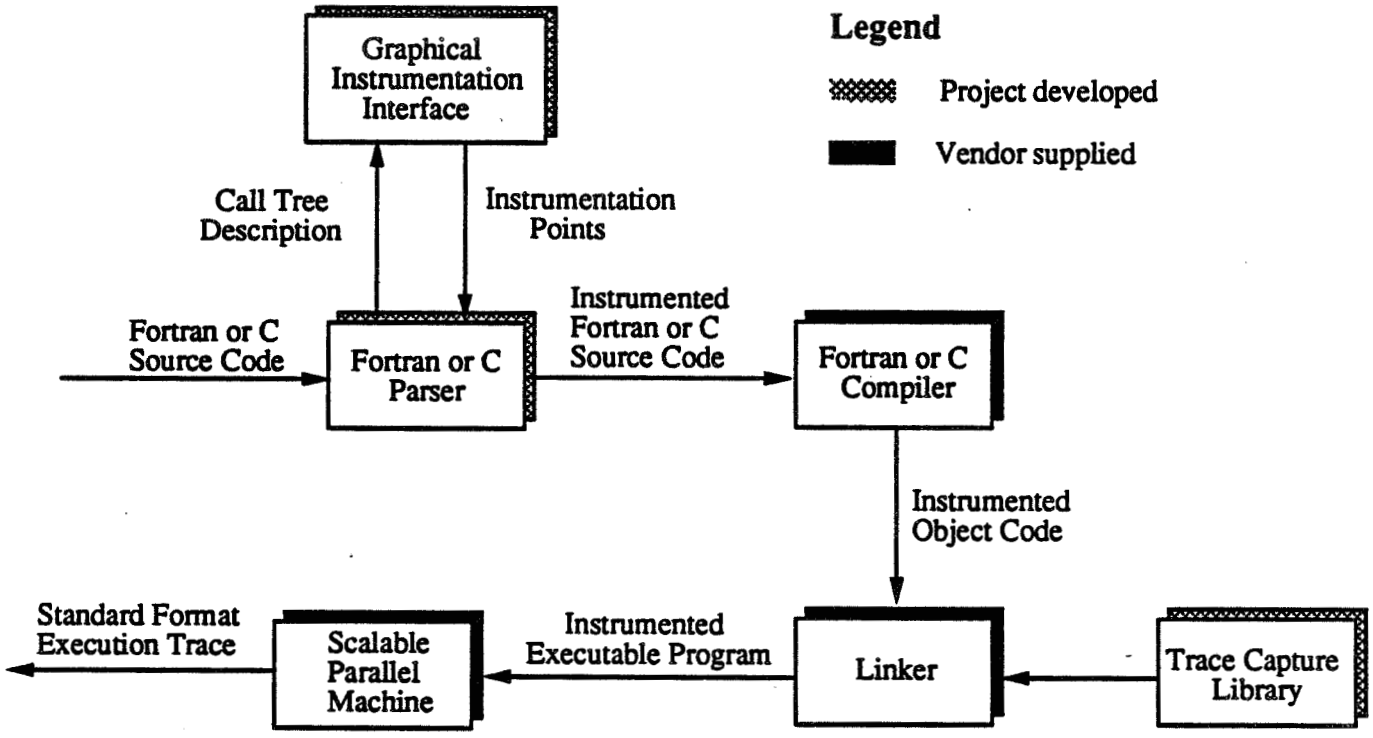
Pablo Performance Data Meta-format

- Embedding performance data semantics in the trace
  - requires software with knowledge of those semantics
  - implies a different trace format for each architecture
  - precludes development of a portable performance analysis environmen
- Pablo data meta-format specifies only performance data record
  - structure
  - component item types (e.g., integer/real or scalar/array)
  - item sizes
  - textual descriptions
- A Pablo trace contains
  - a set of record definitions
  - a stream of (record tag)/(data record) pairs
- Each tag identifies the type of the record to follow

Performance Data Descriptors

```
/*
* "The following records describe a portion
*  of the data that might be recorded on a
*  message passing system with message semantics
*  similar to the Intel iPSC/2."
*/;;

#1: "MessageSend" {
        // "Description" "The following record describes
                          a message transmission event."

        // "PENumber"          "Processor number"
        int                    "PENumber";

        // "Timestamp"         "Time that message was sent"
        int                    "Timestamp";

        // "MessageLength"     "Message size in bytes"
        int                    "MessageLength";

        // "MessageType"       "User defined message type"
        int                    "MessageType";

        // "DestinationCount" "Number of destinations"
        int                    "DestinationCount";

        // "BroadcastList"     "Message destinations""
        int                    "BroadcastList"[];
};;
```

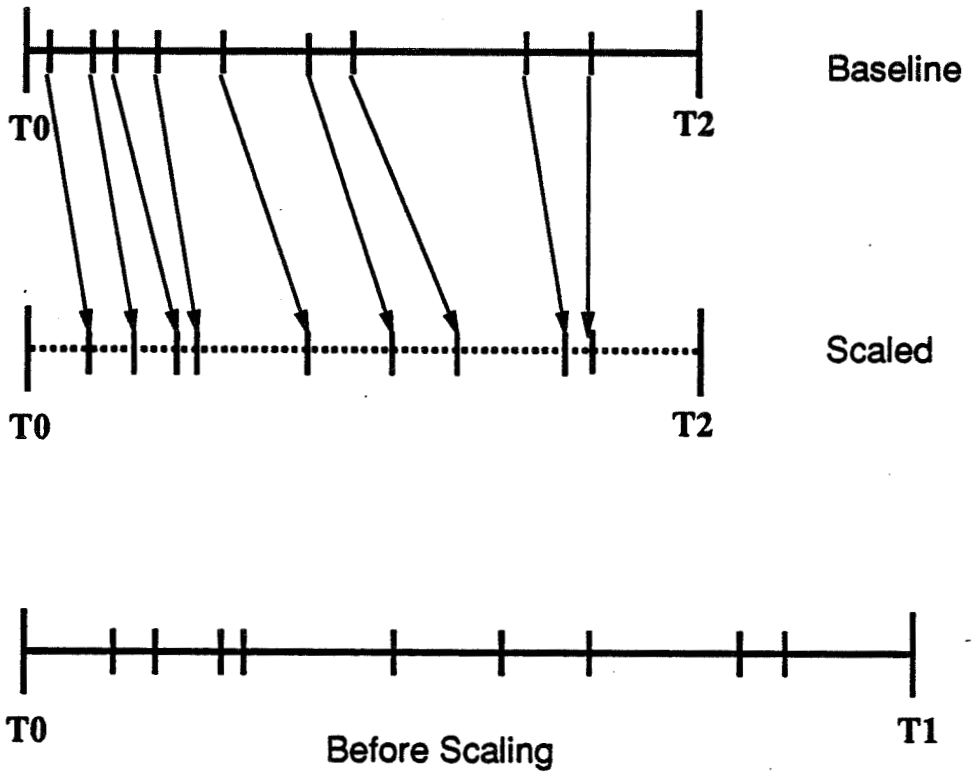

Tapestry



Initial Architectural Targets

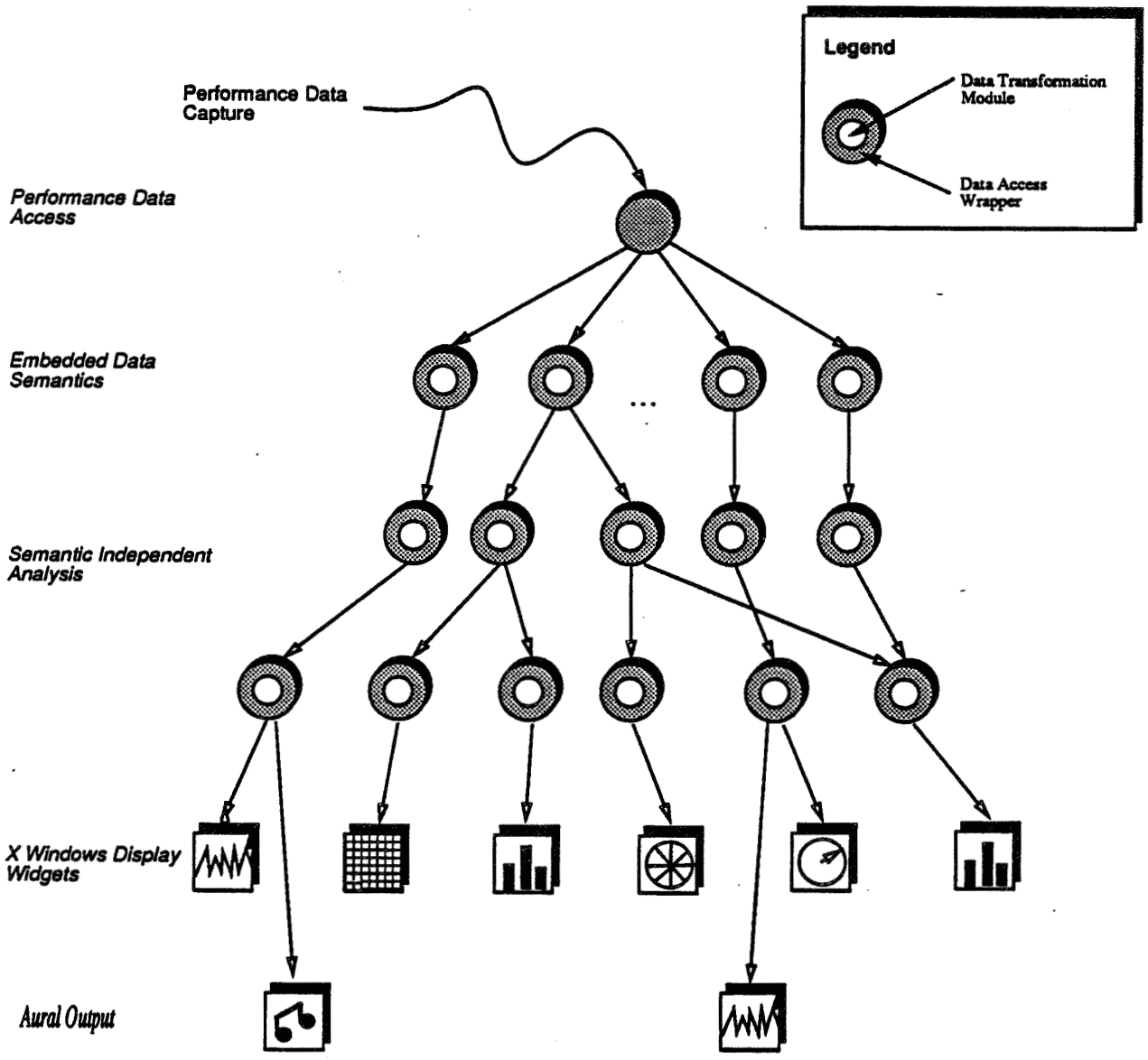
- Thinking Machines CM-2 and decendants
- Intel iPSC/860 and Touchstone

Cross-Architecture Performance Analysis

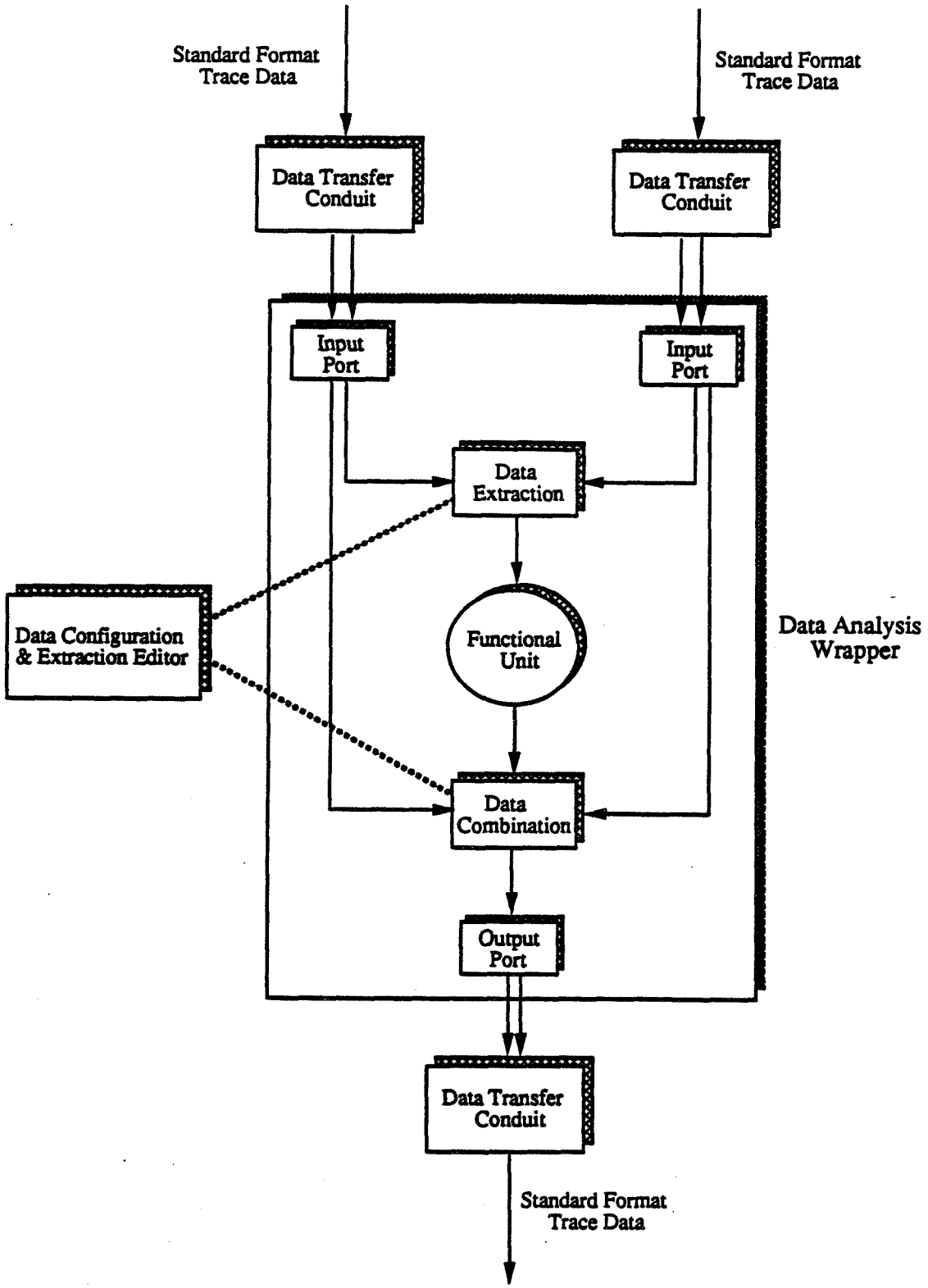


- Total event order for sequential systems
- Partial event order for parallel systems
- Relative event ΔT is complex
- Variations reveal architecture/system software/application interactions

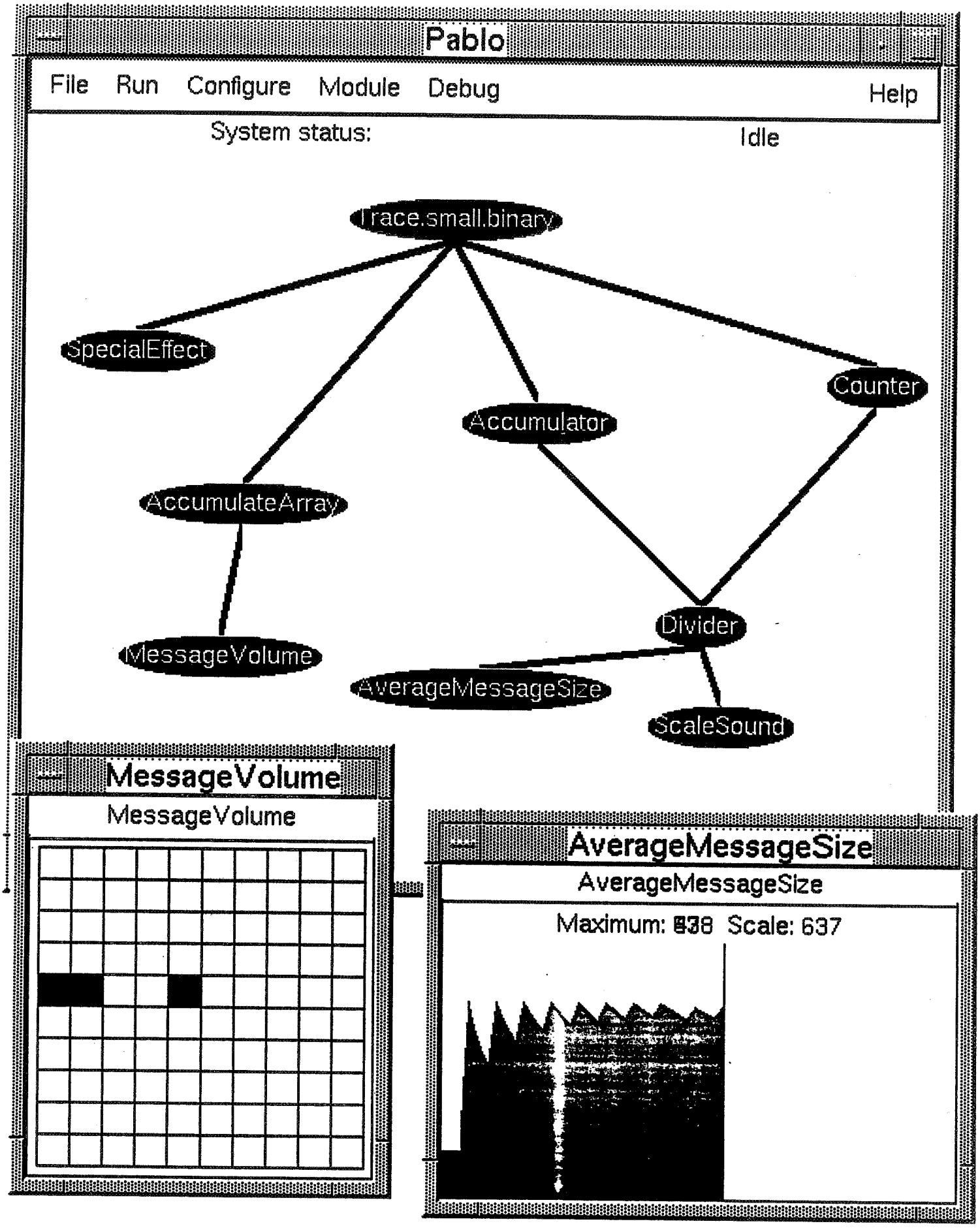
Visualization Environment Infrastructure

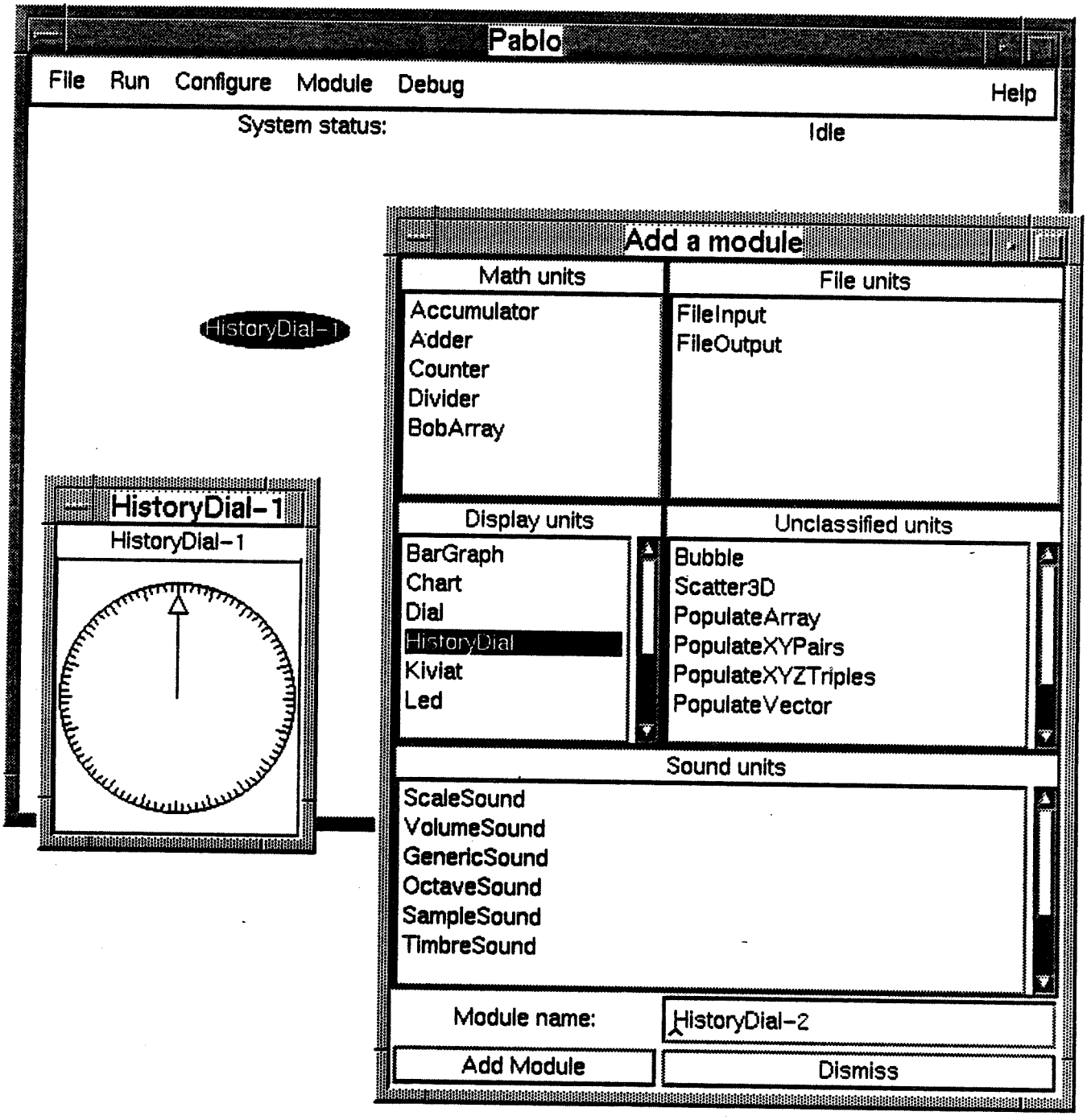


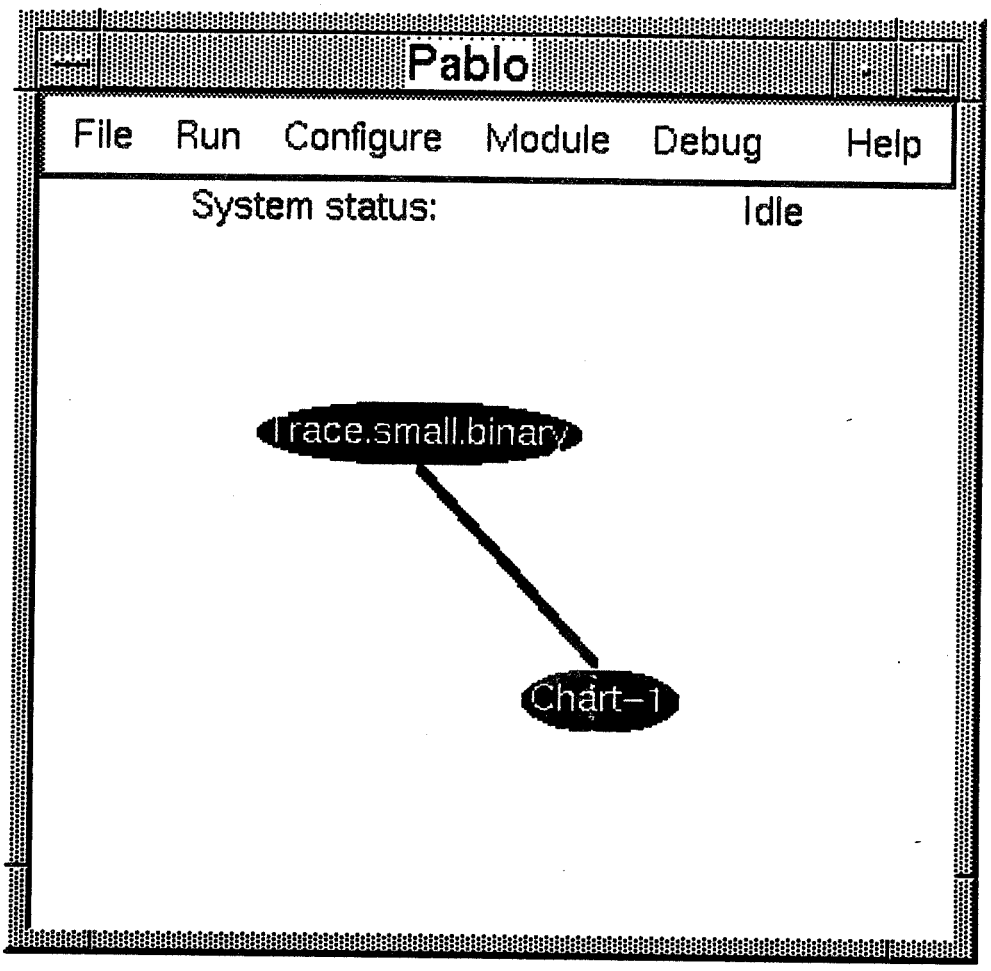
Analysis Module Structure



Tapestry


Pablo
File
Run
Configure
Module
Debug
Help
System status:
Idle
Trace.small.binary
SpecialEffect
Counter
Accumulator
AccumulateArray
Divider
MessageVolume
AverageMessageSize
ScaleSound
MessageVolume
MessageVolume
AverageMessageSize
AverageMessageSize
Maximum: 638 Scale: 637


Pablo
File
Run
Configure
Module
Debug
Help
System status:
Idle
HistoryDial-1
HistoryDial-1
HistoryDial-1
Add a module
Math units
Accumulator
Adder
Counter
Divider
BobArray
File units
FileInput
FileOutput
Display units
BarGraph
Chart
Dial
HistoryDial
Kiviat
Led
Unclassified units
Bubble
Scatter3D
PopulateArray
PopulateXYPairs
PopulateXYZTriples
PopulateVector
Sound units
ScaleSound
VolumeSound
GenericSound
OctaveSound
SampleSound
TimbreSound
Module name:
HistoryDial-2
Add Module
Dismiss


Pablo
File
Run
Configure
Module
Debug
Help
System status:
Idle
Trace.small.binary
Chart-1

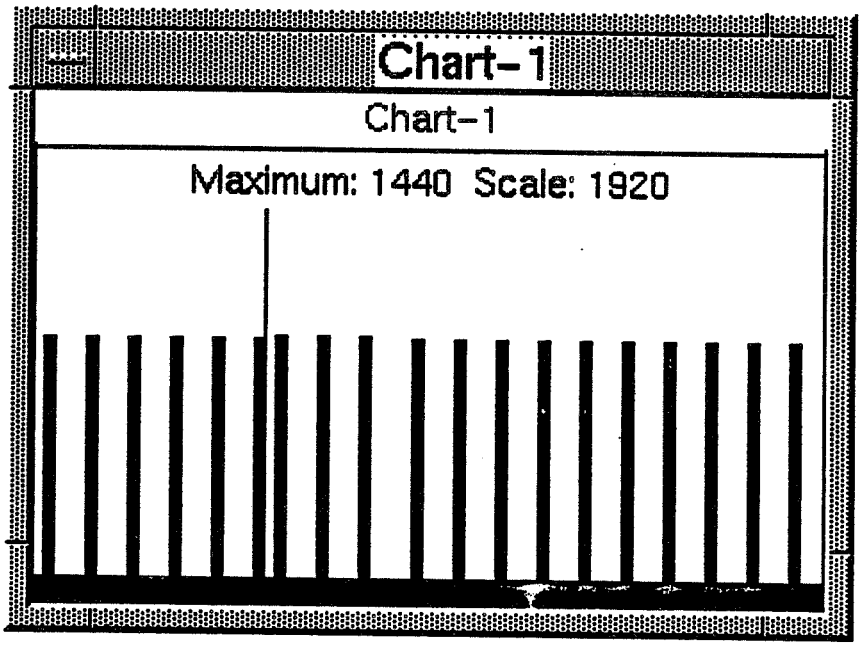


Chart-1
Chart-1
Maximum: 1440 Scale: 1920

Visualization Displays

- Operational
  - bargraph
  - bubble chart
  - strip chart
  - contour plot
  - dial
  - gantt chart
  - graph layout and display
  - history dial
  - interval plot
  - kiviat diagram
  - LED array (discrete bargraph)
  - X-Y line/scatter plot
  - matrix display
  - pie chart
  - polar plot
  - 3-dimensional scatter plot

- Proposed/Planned
  - stereoscopic 3-dimensional displays
  - scatter plot matrices
  - Chernoff faces
  - Feynman diagrams

Performance Scalability

- Parallelism
  - data capture
  - data reduction
- Hardware support
  - timestamps
  - data capture
- Data presentation
  - system gestalt
  - equivalence classes
  - phases
  - detail for processor subsets

Performance Data Estimates

- Assumption: 1024 nodes

- Processor utilization
  - Timestamp – 4 bytes
  - Utilization – 4 bytes
  - 250 millisecond samples

  $\Rightarrow$ 16K bytes/second (57 MB/hour)

- Message traffic
  - Timestamp – 4 bytes
  - Source – 8 bytes (node and process)
  - Destination – 8 bytes (node and process)
  - Message size – 4 bytes
  - Message type – 4 bytes
  - 100 millisecond intertransmission time

  $\Rightarrow$ 280K bytes/second (1 GB/hour)

Real-time data reduction is essential

Pablo Performance Data Sonification

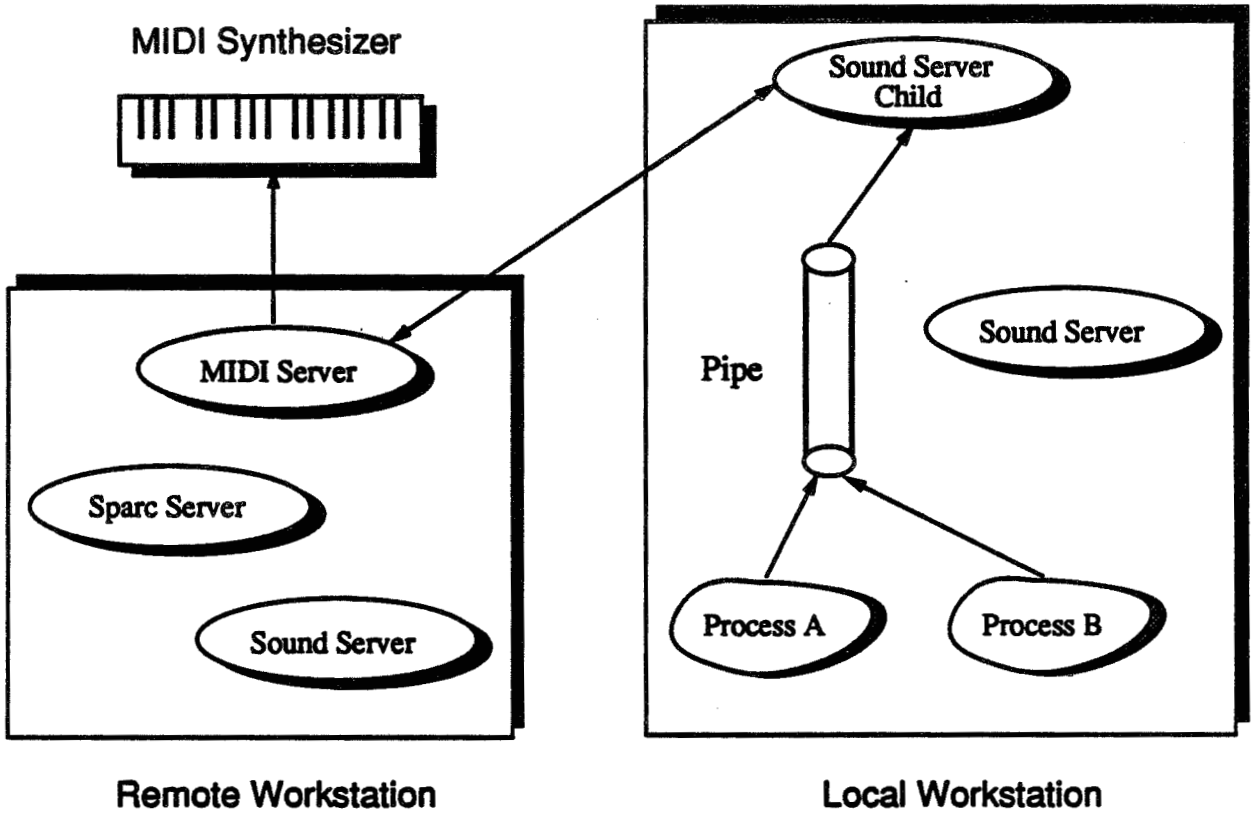
- Psychological studies show that some types of data are more easily perceived aurally rather than visually
- Rich set of sound attributes
  - pitch
  - loudness
  - attack
  - decay
  - balance
  - duration
  - timbre
- Allow users to choose the best mix of sound and graphics
- Sonic widgets
  - spoken warnings
  - alarms
  - data-defined mappings
- Two implementations
  - SparcStation audio — low fidelity but widely available
  - Musical Instrument Digital Interface (MIDI) — sound synthesis

Sonification Design Goals

- Selection of *remote* sound hardware
- Hardware independence
- Real-time sound generation
- Keyboard sound metaphor
- Abstractions to support "sonic widgets"

⇒ Server/client architecture

Sound Server Structure



- Initiates connections to host/port (`/etc/services`)
- Returns a list of currently open connections
- Describes hardware sound resources
- Three description levels
  - message configuration
  - parameter defaults
  - device parameters

Tapestry

Writing Sonic Widgets

- Two widget types:
  - real-time
  - historical
- Sound control file — atomic unit of audio presentation
- Control file contains messages with three argument types:
  - fixed
  - configurable at widget creation
  - data dependent

Future Work

- Module synchronization
- Expanded analysis module library
- Help facility
- Expanded graphical displays
- Parallelization

Software Designers/Developers

- Pablo is the effort of many people

- Special thanks to:
  - Bob Olson
  - Ruth Aydt
  - Tara Madhyastha
  - Brian Totty
  - Tom Birkett
  - Bobby Nazief
  - Ted Nelson
  - Al Malony

The Choices Object-Oriented Operating Systems

ICLASS Review

David K. Raila, Lee Lup Yuen

Department of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield Ave., Urbana, IL 61801-2987

1. Kernel Progress
2. Applications and Application Support
3. Future Directions
4. Conclusions

1. Kernel Progress

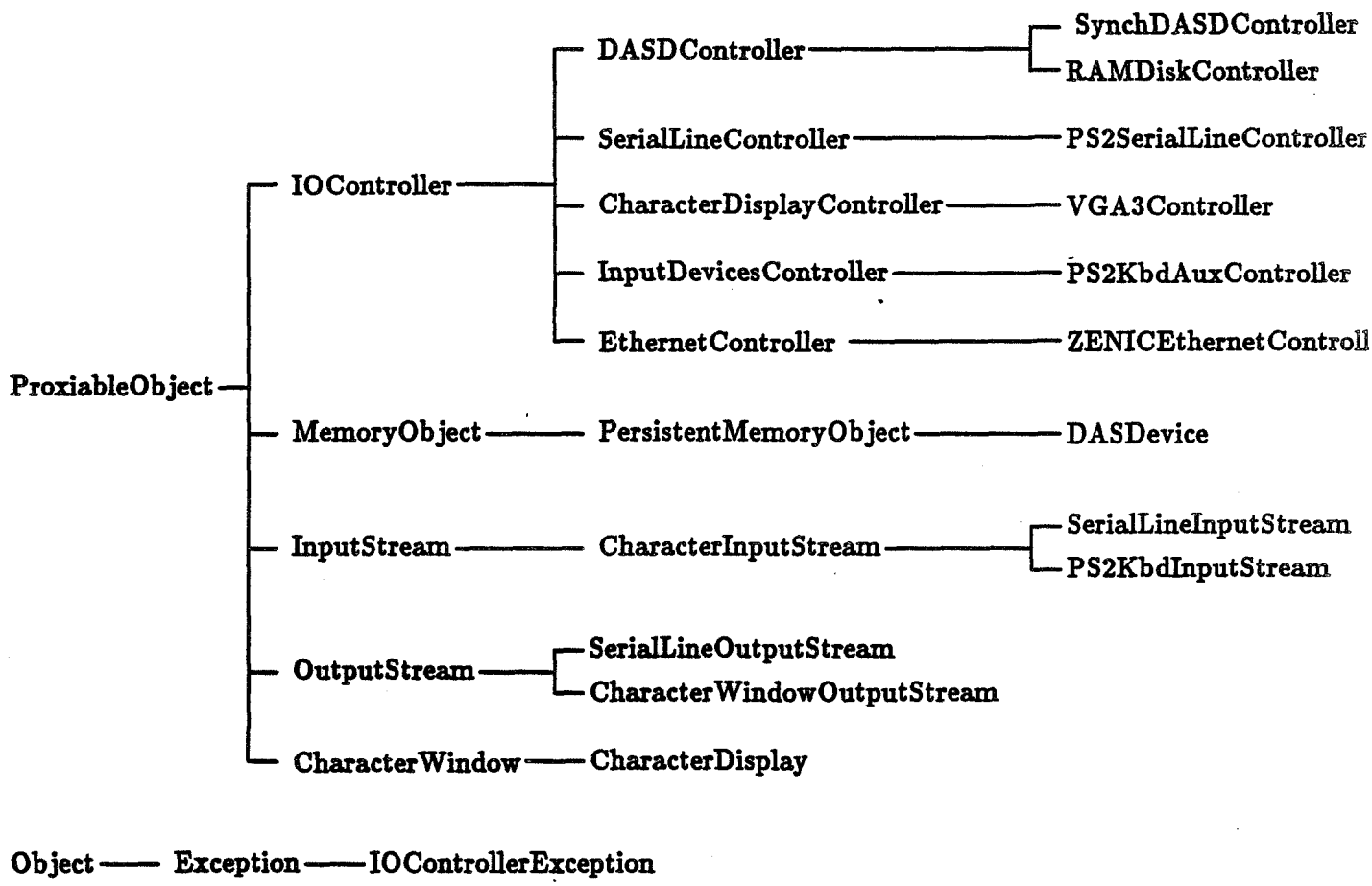
- Networking
  - C++ TCP/IP implementation in use
  - Networking utilities on Choices (ftp, telnet, etc.)
  - Remote Procedure Call
    * Remote method calls to Choices objects
    * Implemented on TCP/IP
    * Graphical interface to manipulate Choices objects
- Protection Rings
  - Multiple levels of protection beyond Kernel and User
  - Inner rings protected from outer rings
  - Services can move outside Kernel into intermediate ring
  - Simplifies debugging subsystems
  - Supports development of Choices microkernel

Kernel Progress Continued

- Filesystems
  - Portability
  - Persistent Objects
  - Record based, ISAM, Log, tar
- Virtual Memory
  - New implementation
  - Additional Features implemented efficiently
    * Copy on write
    * Distributed virtual memory
- Device driver framework
  - Location of devices and drivers
  - Dynamic loading of drivers
  - Controllers support common actions
  - Superclasses encapsulate common machine-independen characteristics
  - Machine and device dependent code minimized

Kernel Progress Continued

- Driver/Controller Class Hierarchy



Kernel Progress Continued

- Portability
  - SPARCstation port completed
  - IBM PS/2 port completed
  - Hypercube port in progress
  - SGI port in progress

2. Applications and Application Support

Application Development Environment

- Choices running full-time on multiprocessor target
  - Multiple users running applications simultaneously
  - ftp, telnet, and common shell utilities for support
  - No need for Application developers to be kernel gurus
  - Feedback from users of real applications
- Choices and applications instrumented for performance analysis
  - Instrumented Kernel and applications to locate problems
  - Instrumentation data used with Pablo visualization system

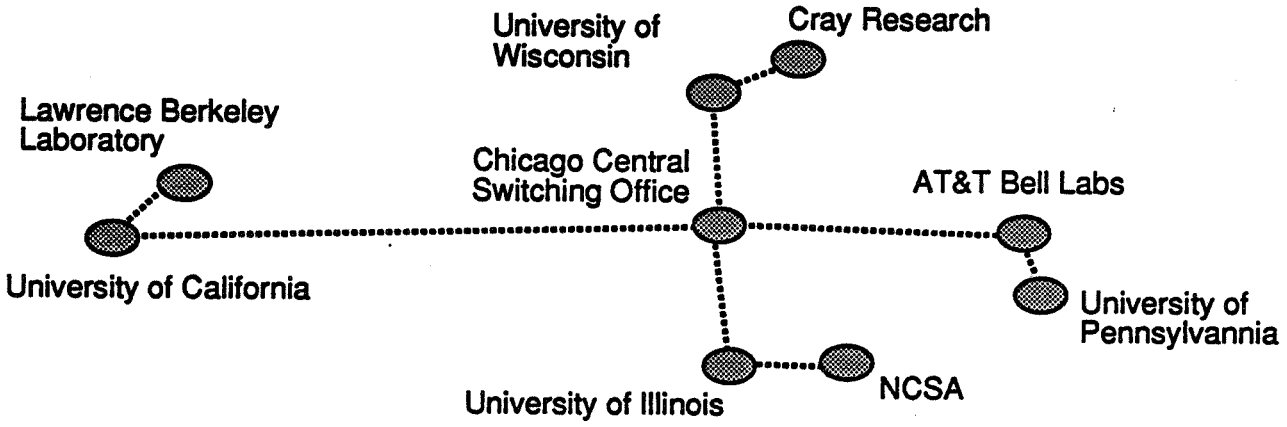
Unix Compatibility Support

- Goals
  - Source/Object code compatibility with Unix
  - Ease the migration path from Unix to Choices
  - Allow testing of Choices with known applications
  - Self-hosting Choices
- Implementations
  - Binary Compatible System Call Interface
    - Invoke Choices services in response to requests from Unix processes for supported services
    - Emulate Unix operations not directly supported
    - Encapsulate different varieties of Unix in classes to support different versions of Unix
  - Binary Compatibility Library
    - System calls implemented in user libraries
    - Object code linked with libraries for Choices application

VirtualChoices

- Choices running on Unix abstract machine
- Choices grafted onto a C++ tasking package
- Environment for developing machine-independent code
- Runs under Unix
  - Choices code used *without* change
  - No need for access to a target for booting
  - Fast "test-edit-compile-retest" cycle, no rebooting machines
  - Source level debugging
- What is the minimal Choices abstract machine necessary to support higher level Choices Kernels?

Archos Xunet II Switch Controller



- XUNET II is 45 Mbits/s WAN
- XUNET III will have 600 Mbit/s lines
- Virtual circuit/ATM based.
- Switch control computer runs *Choices* as a custom embedded system for call processing.

  Switch controller must:

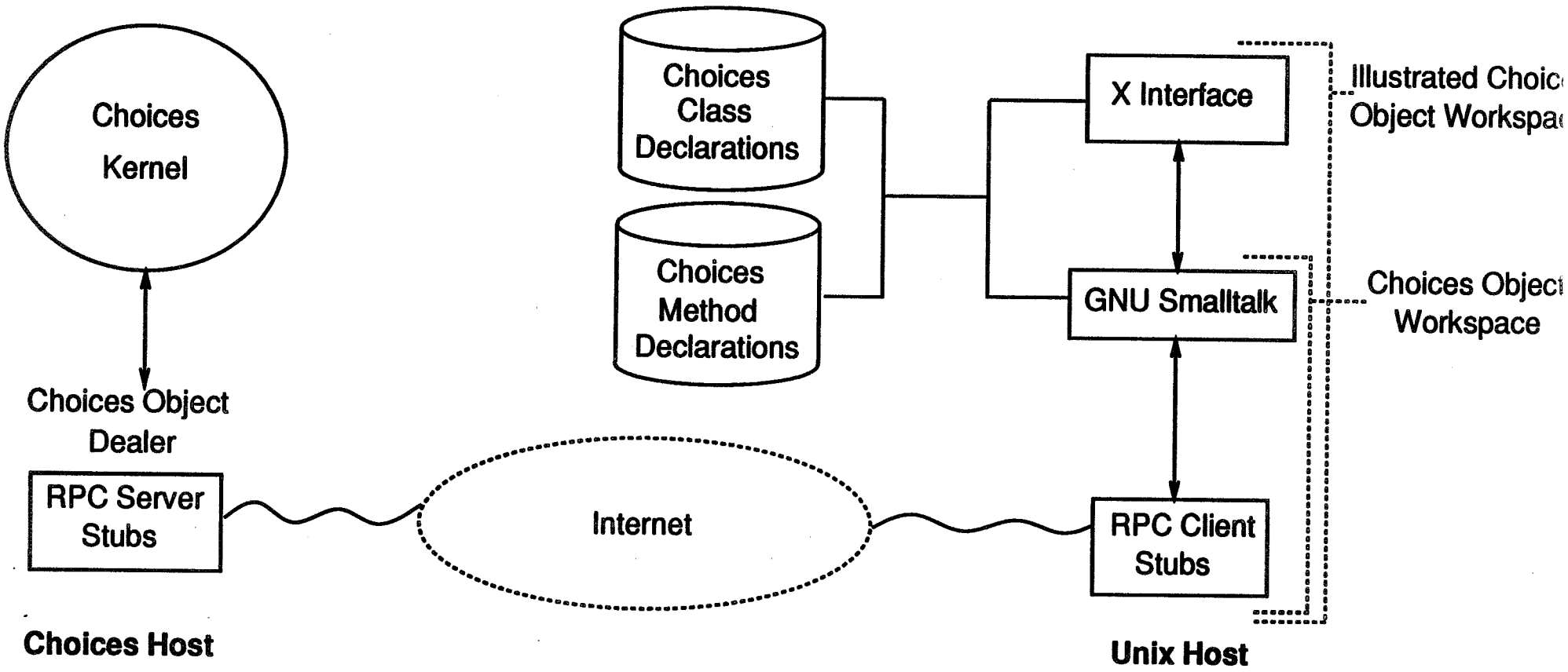
  - provide wide variety of user and network services
  - manage call setup/takedown in a large virtual circuit space
  - be portable, extensible, scalable

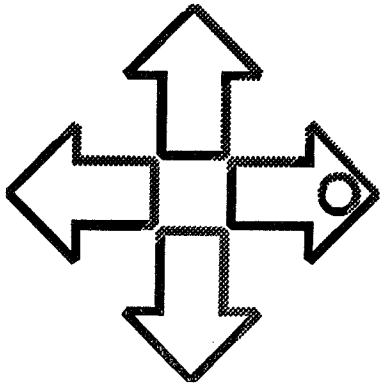
3. Future Directions

- Choices Microkernel
  - Filesystem, networking, etc. moved out of kernel
  - Move policies outside of kernel
  - Support for exception handling
- Object Oriented Distributed Systems
  - Object oriented IPC/RPC investigation
  - Object oriented support for distribution and reliability
  - Hypercube port
- Performance analysis
  - Visualization using Pablo system
  - Application and Kernel analysis
- Miscellaneous
  - Choices port to massive multiprocessor??
  - Real time systems
  - Collaboration with USL

4. Conclusion

- Choices provides and ideal O/S research testbed
  - Object Oriented model supports rapid prototyping
  - Reusability — inheriting common code
  - Extensibility — subclassing abstract classes
  - Portability — encapsulating machine-dependent code
  - Efficiency — performance as good or better than vendor operating systems
- Choices provides parallel object-oriented application testbed
  - Object oriented system interface
  - Performance analysis with Pablo system
  - Support for parallel and message passing applications
  - Real applications
    * Archos
    * ICOW



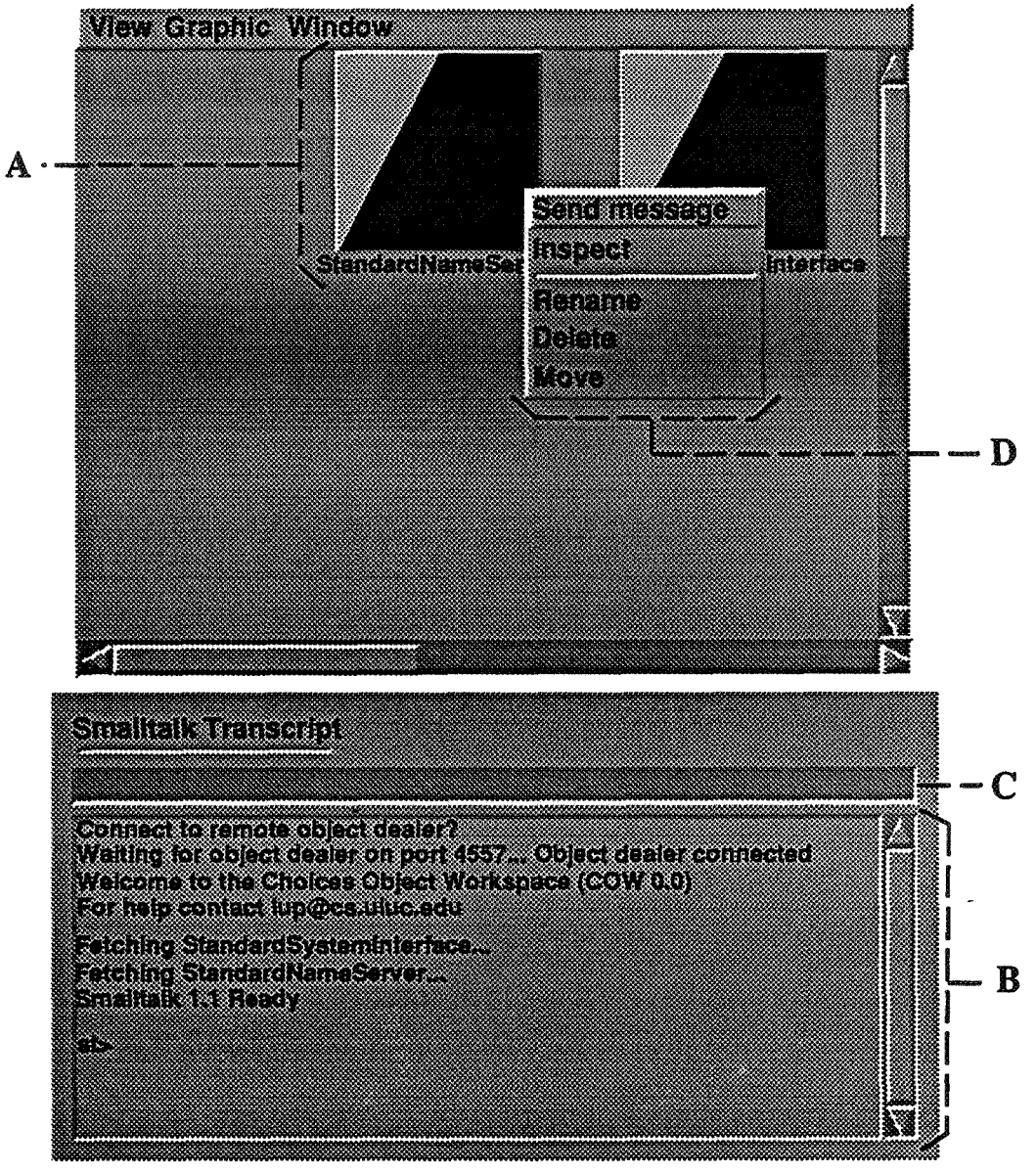


Illustrated Choices Object Workspace

Graphical interface for working with objects residing in the object–oriented Choices kernel

Users can send messages to Choices objects interactively and can also inspect instances of classes

Graphical interface runs under UNIX and communicates with a remote Choices host over the network



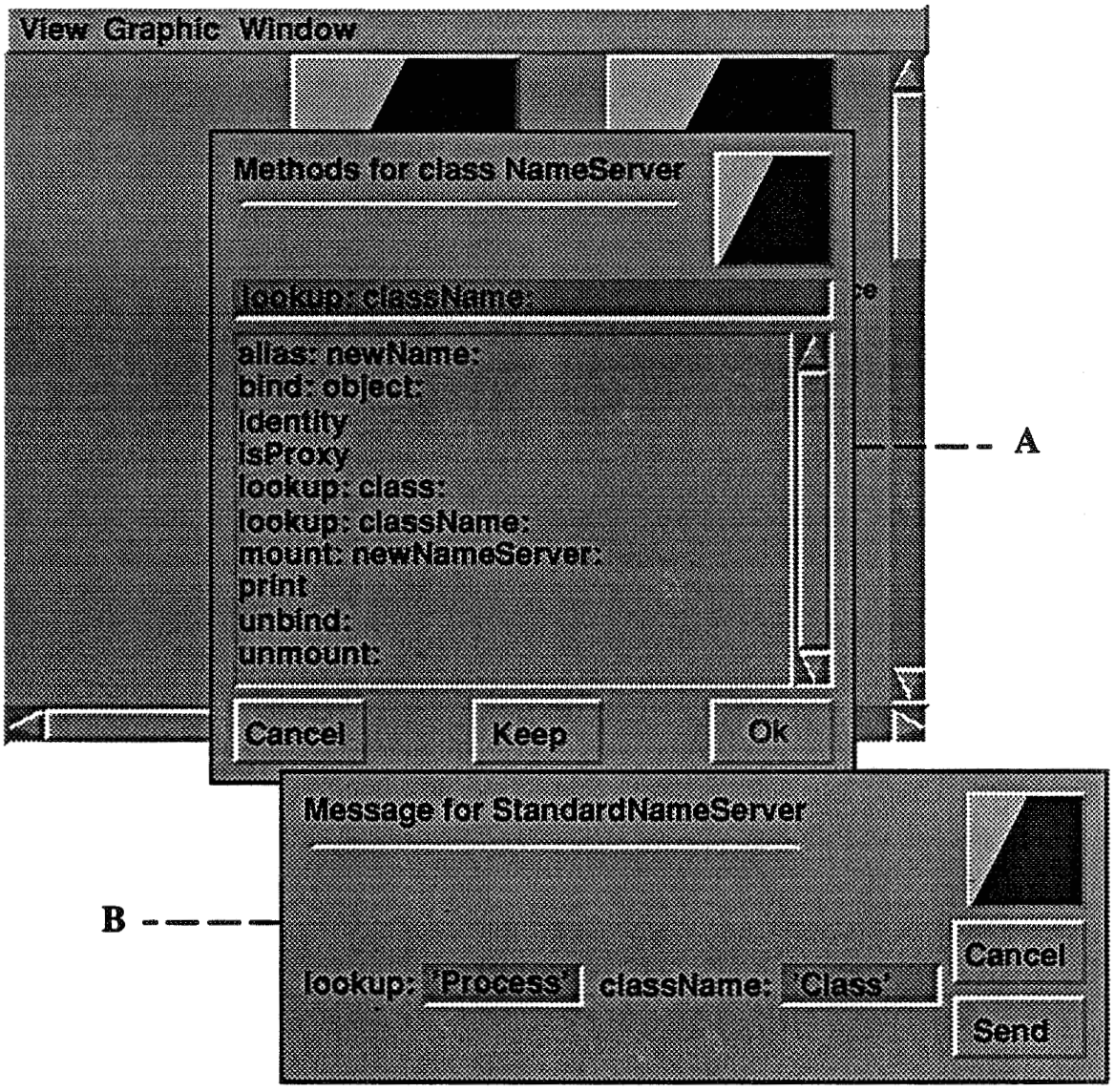
A. This icon represents the StandardNameServer object in the Choices system. The StandardNameServer is used for fetching user–accessible Choices objects, like StandardInput, StandardOutput and FileSystemInterface.

B. The graphical interface (written with the InterViews 3.0 toolkit from Stanford University) emits Smalltalk commands to perform operations on Choices objects. The Smalltalk Transcript window shows the execution of the Smalltalk commands. (The Smalltalk interpreter used is GNU Smalltalk 1.1 from Free Software Foundation.)

C. Users may enter Smalltalk commands here for execution by the Smalltalk interpreter.

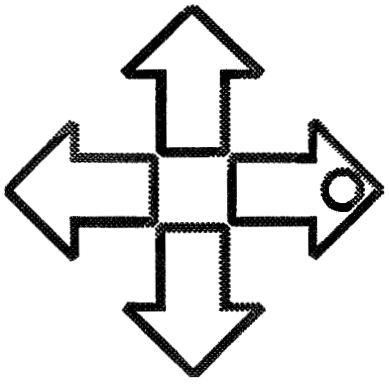
D. This menu pops up when we press the mouse button on the StandardNameServer icon. To fetch an object from the StandardNameServer, we have to send it a message. (Proceed to next slide...)

Illustrated
Choices
Object Workspace

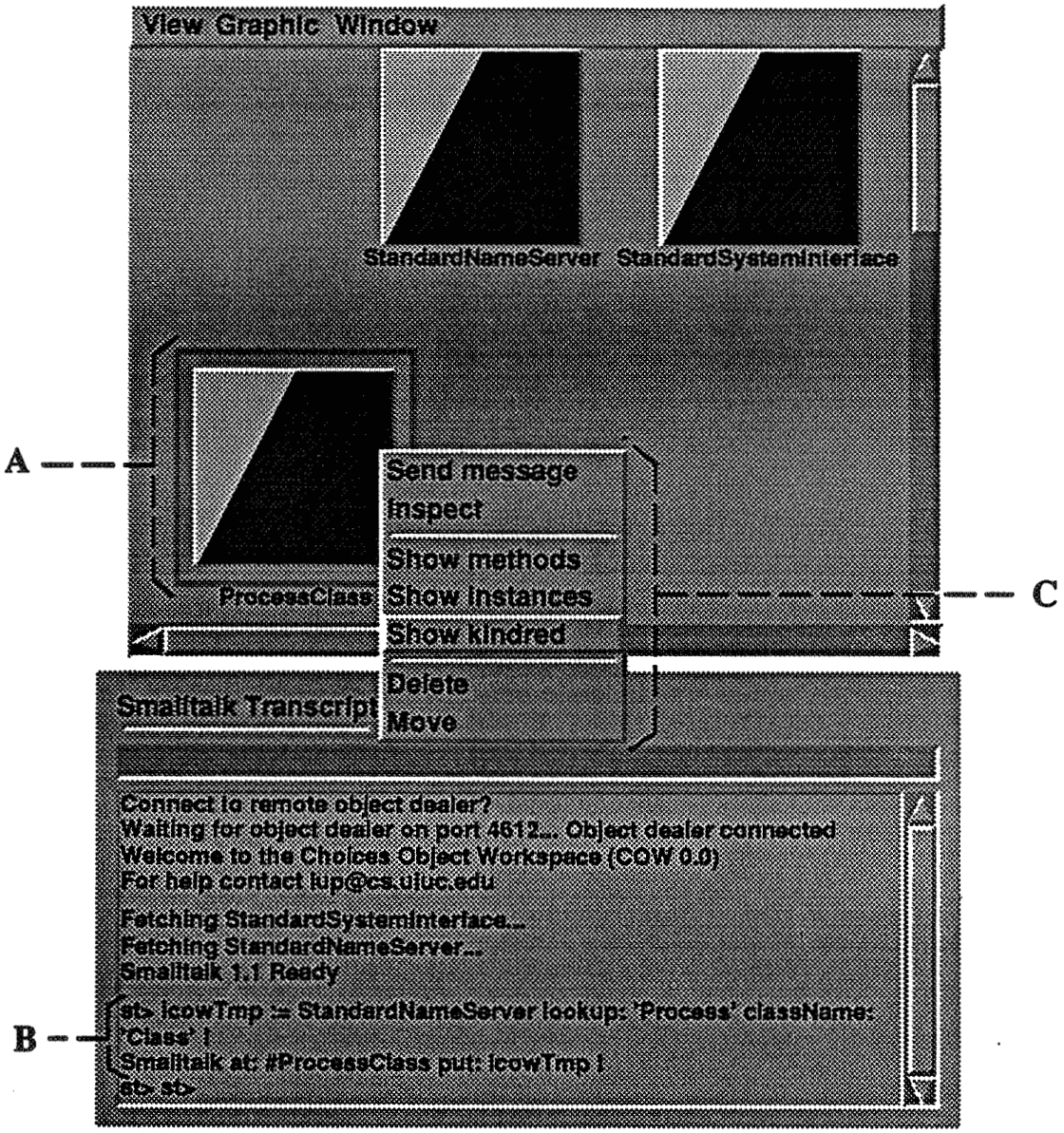


A. This method selector box appears after selecting the "Send message" option. We are shown all of the StandardNameServer's methods that can be used as the basis of the message. We select the "lookup: className:" method and click "Ok", which brings up the message box for that method.

B. In this message box, we fill in the arguments for the method. In this message we indicate that we want to fetch the class Process from the StandardNameServer. After filling in the arguments, we click "Send". (Proceed to next slide...)



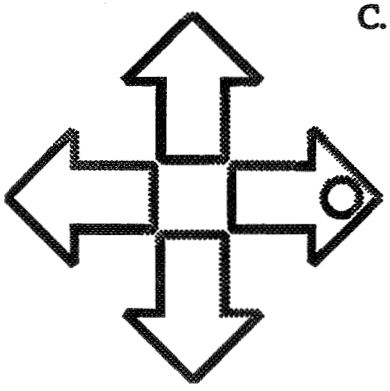
Illustrated
Choices
Object Workspace



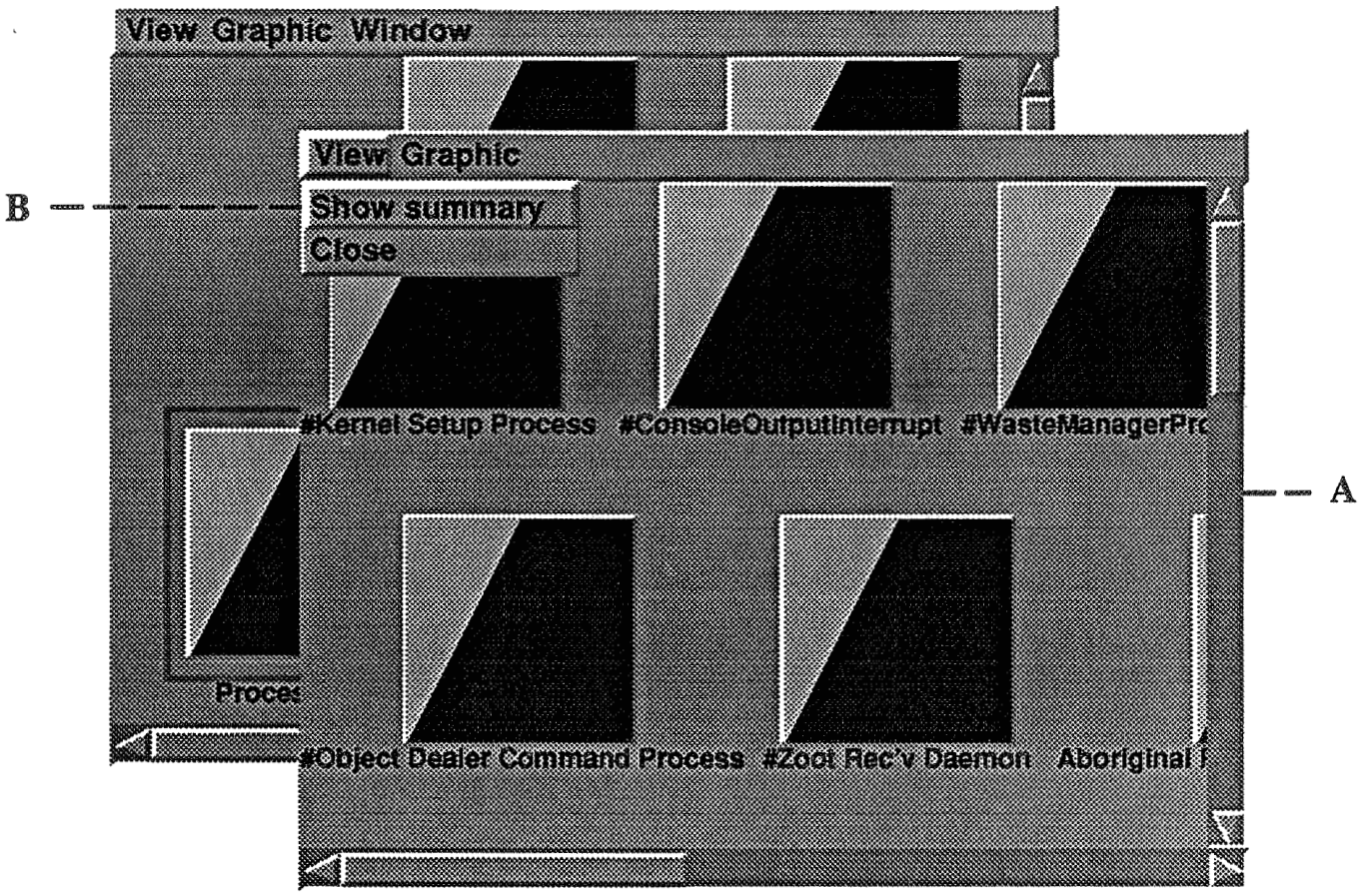
A. The Process class is returned by the StandardNameServer in the Choices system and is displayed as an icon.

B. The Smalltalk Transcript window shows the Smalltalk commands that were executed while processing the message. There is a Smalltalk object named "StandardNameServer" that is called to handle the message. That Smalltalk object forwards the message to the real StandardNameServer on a remote Choices system. The real object then returns the result to the Smalltalk interpreter, which displays the ProcessClass icon.

C. We can now display all instances of class Process and its subclasses. We press the mouse button on the icon and select "Show kindred". (Proceed to next slide...)

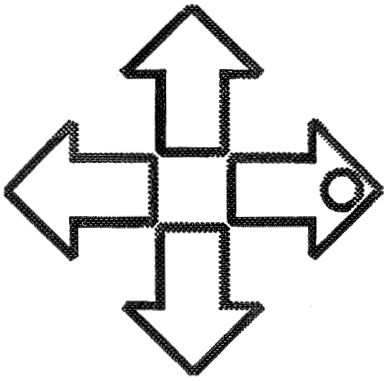


Illustrated
Choices
Object Workspace

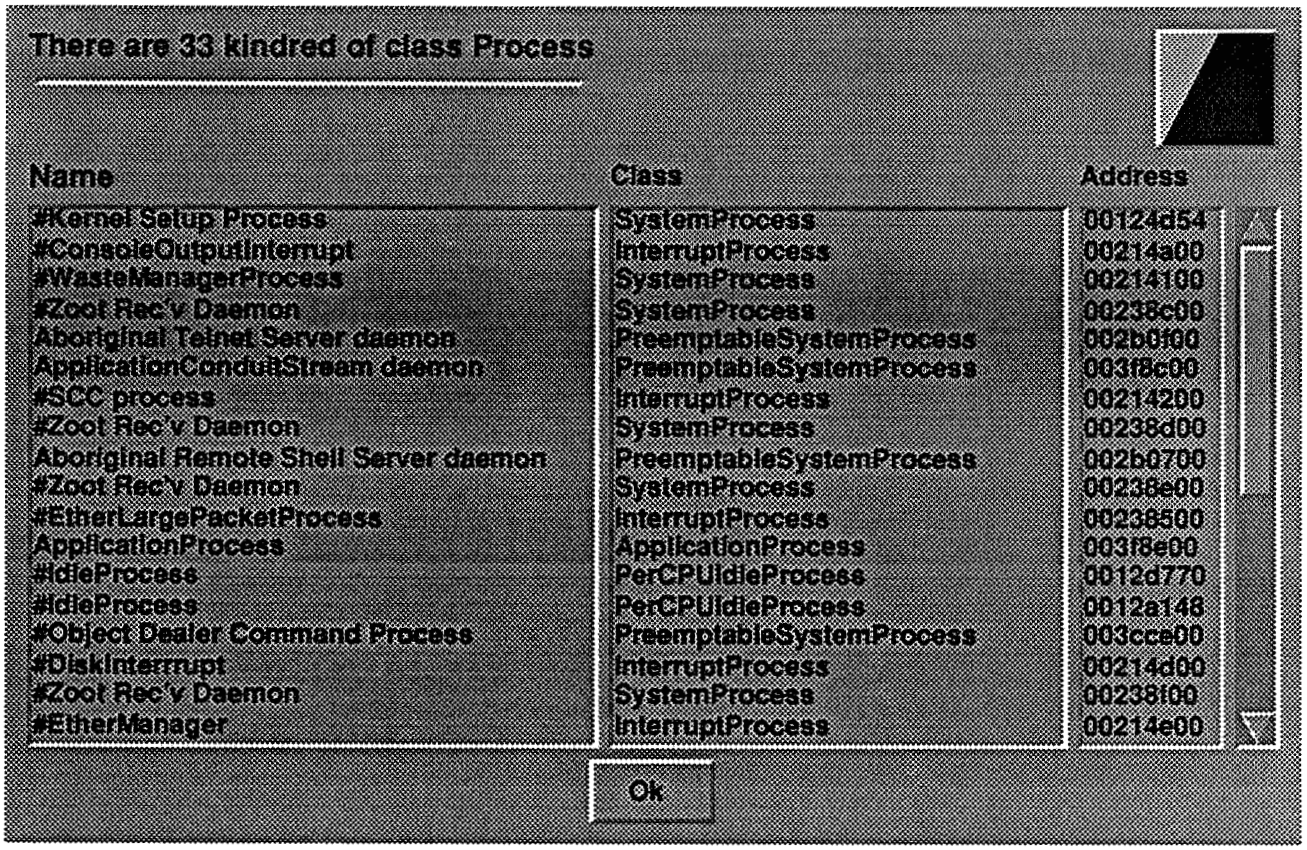


A. A window appears, displaying all instances of class Process and its subclasses. Each instance represents an individual process that is currently active in the Choices system.

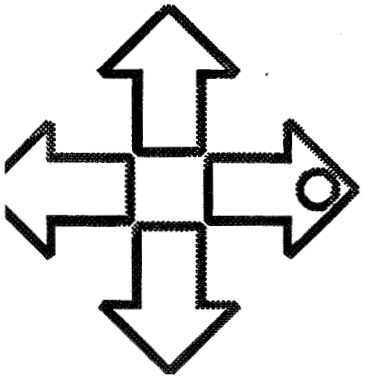
B. To see information about the instances in text, we choose "Show summary" from the menubar. (Proceed to next slide...)



Illustrated
Choices
Object Workspace



Here we see the names, classes and memory addresses of the active processes. There are networking daemons (Telnet, Remote Shell, Zoot Receiver), I/O interrupt–handling processes (Console Output, SCC Process, Disk Interrupt, Ether Manager) and an application process.



Illustrated
Choices
Object Workspace

Constraint Processing in Logic Programming and Automated Deduction

Alan M. Frisch

Department of Computer Science
and
Beckman Institute
University of Illinois

Introduction

Efficient deduction: build in special-purpose methods

One way to do this: Build in constraints

- build constraints into language
- build constraint handling into reasoner
- analyze result with substitutional framework

Constraints can be built into wide range of deductive systems

- Planners and Schedulers.
- Language parsers and generators, particularly those based on feature structures.
- Logic programming systems.
- General deductive systems for problem solving, common-sense reasoning, mathematical theorem proving.
- Knowledge retrieval.
- Deductive data bases.
- Equational rewriting.

Basic Idea

- Many systems use unification to handle variables
  - Known: $member(y, cons(y, x))$
  - Query: $member(z, cons(2, nil))$
  - unification matches 2 to y, 2 to z, and nil to x

- Unification is form of constraint solving
  - solve equation
    $member(y, cons(y, x)) = member(z, cons(2, nil))$
  - yields $\{y = 2, z = 2, x = nil\}$

- Generalize unification to arbitrary constraint solving
  - Known: $P(x, y)/x + y = 2$
  - Query: $P(1, y')/y'\ is\ Odd$
  - Solve: $\{x = 1,\ y = y',\ x + y = 2,\ y'\ is\ Odd\}$
  - Solution: yes, $x = y = y' = 1$

New Domains

Can be thought of as building in domains and relations

ɜtem	Domains	Relations
ɔlog	symbolic terms (i.e., finite trees)	$=$
ɔlog II	infinite trees	$= \neq$
ɔlog III	boolean terms rationals w/ $+$, $*$ infinite trees lists w/ concatenation	$= \neq$ $= \neq \leq \geq$ $= \neq$ $= \neq$
P(R)	reals with $+$, $-$, $*$	$= \leq \geq$
ƆPC	symbolic terms	arbitrary monadic predicates (i.e., categories)
ɪstraint ʒic	symbolic terms	arbitrary predicates

But Prolog has Arithmetic!

- Prolog: x is $y + z$

- Need ability to unify $x + 1$ and 3
  (i.e. to solve $x + 1 = 3$)

- Example: $Fib(n, k)$ n^{th} Fibonacci is k
  - 1) find k given n
    2) find n given k
  - Prolog does (1) but not (2)
  - Constraint deduction (with $+ -$ $\leq$) does both

Advantages

Expressiveness

- problem can be encoded with fewer and shorter formulas
- implicit answers can be given—in some cases an implicit answer is finite where an explicit answer would be infinite

Efficiency

- Walther's [1985] solution to Schubert's Steamroller problem
- Frisch's work on deductive parsing [1986] and deductive database problem [1990]

Substitutional Framework

- For Sorted First-Order Predicate Calculus [Frisch, 1989]
  - build sorts into any deductive system based on unification
  - constraints solved by sorted unification
  - Universal Sorted Unification Algorithm [Frisch & Cohn, 1990]

- For Constraints on Multiple Variables

- Main Idea:
  - $P(x)$ is schematic for $\{P(a), P(b), P(c), \ldots\}$
  - $P(x)/C(x)$ is schematic for $\{P(a), P(b), P(c) \ldots\}$

Results

- A systematic method to transform an ordinary deduction system to a constraint deduction system

- Theoretical underpinnings of all systems constructed in this manner.
  - Conditions that are necessary and sufficient for this to work.

Applications

- Parsing grammars with Feature Co-occurrence Restriction
- Deduction in Modal Logics
- Sorts in FRAPPS - A toolkit for building resolution-based systems

Current and Future Directions

General algorithm for sorted unification (with Tony Cohn)

Special cases when sorted unification is decidable (with Tomas Uribe)

Modal deduction with equality for range of modal logics (with Rich Scherl)

Constraints in Planning (Herry Sutanto)

Induction in constraint logic (with C. David Page)

Extend framework to handle equational constraints, thus subsuming theory of constraint logic programming

Space of hybrid reasoners (AAAI Fall Symposium)

Spectral Methods for Meteorology

on

Advanced Computer Architectures

Faisal Saied

Department of Computer Science

University of Illinois at Urbana-Champaign

ICLASS Review

October 18, 1991

Acknowledgements

Steve Cote

Michael Ham

Overview

- Solution Techniques for Partial Differential Equations (PDE's)
- Spectral Methods for PDEs
- Shallow water equations for a rotating sphere
- Spherical, Rectangular Grids
- Expansion in Spherical Harmonics
- Transforms for the Spherical Harmonic Expansion
- Advanced Computer Architectures
- Parallel Performance of a 2D FFT on the Intel iPSC/860
- Planned Implementation on the Connection Machine (CM2)

Solution Techniques for Partial Differential Equations (PDE's)

- Fast Poisson Solvers item Linear Algebra Solvers (Direct, Iterative)
- ADI methods
- Multigrid
- Domain Decomposition
- Spectral Methods
- Explicit vs. Implicit methods for time-dependent problems

Spectral Methods for PDEs

Approximate solution of $\mathcal{L}u = f$ or $u_t = \mathcal{L}u + f$ by a truncated expansion in terms of an appropriate family of basis functions:

$$u(\mathbf{x}, \mathbf{t}) \approx U_N(\mathbf{x}, \mathbf{t}) = \sum_{j=0}^{N} \tilde{U}_j(t) B_j(\mathbf{x})$$

Typical families of basis functions:

- Fourier $e^{ikx}, \sin(kx), \cos(kx)$
- Chebyshev $T_k(x)$
- Legendre $P_k(x)$
- Spherical Harmonics $Y_n^l(x) = P_n^l(x) e^{ilx}$

Key characteristics of spectral methods:

1. Excellent approximation properties ($\Rightarrow$ good resolution with relatively few degrees of freedom)
2. Fast transforms from "physical" space to "transform" space, (usually the FFT)

Shallow water equations for a rotating sphere

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} - fv + g\frac{\partial h}{\partial x} = 0$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + fu + g\frac{\partial h}{\partial y} = 0$$

$$\frac{\partial h}{\partial t} + u\frac{\partial h}{\partial x} + v\frac{\partial h}{\partial y} + h\left(\frac{\partial u}{\partial x} + \frac{\partial u}{\partial x}\right) = 0$$

u, v are the Cartesian components of wind velocity

h is the height of the free surface

g is the acceleration due to gravity

f is the Coriolis force.

Shallow water equations for a rotating sphere

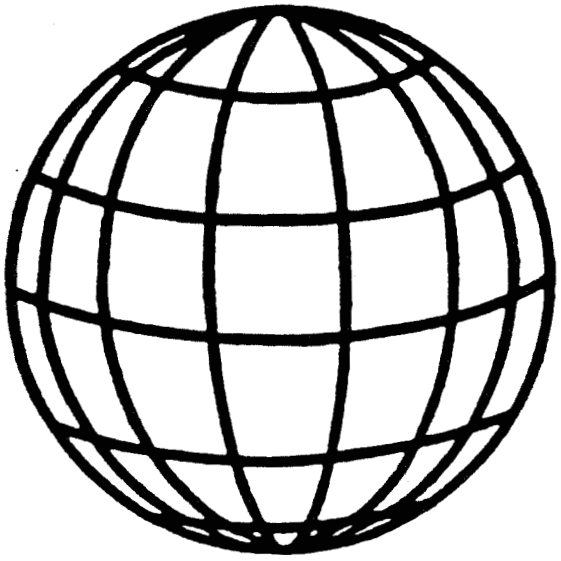
The full Global Spectral Model (GSM) in Meteorology is 3-D, i. e. it has several layers at varying heights, rather than a single layer.

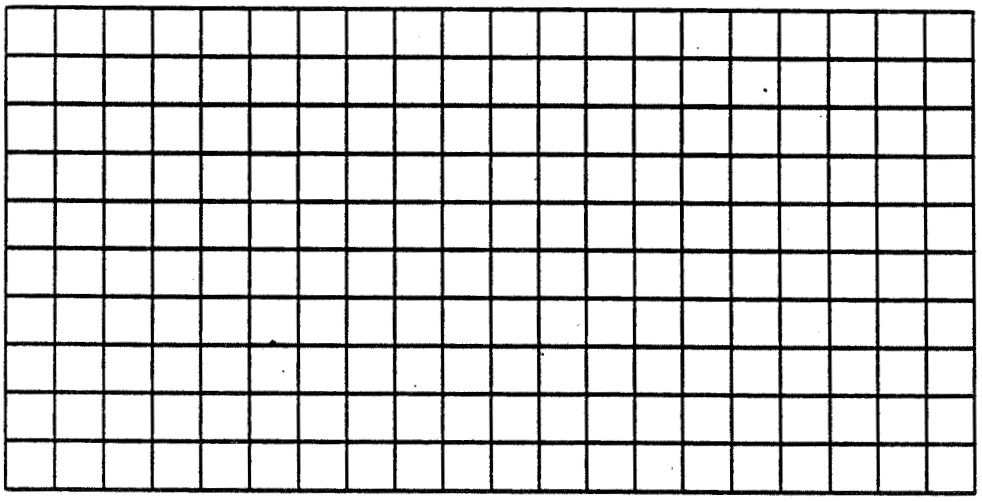
There are numerous physical effects that are added on in the full model.

However, the (2-D) Shallow Water Equations are the simplest subset of equations that capture some non-trivial aspects of the full Global Spectral Models. In particular, the transforms arising from the Spherical Harmonic Expansions are the same, and demonstrating a speed-up for these transforms implies a similar potential speed-up for the GSM.

Variants of the Global Spectral Model are used in Extended Range Forecasting and supply data for Regional Models.

Spherical, Rectangular Grids





Expansion in Spherical Harmonics

Spherical Harmonics:

$$Y_n^l(\phi, \lambda) = P_n^l(\sin(\phi))\ e^{il\lambda}$$

Expansion on F in terms of Spherical Harmonics:

$$F(\phi, \lambda) = \sum_{l=-J}^{J} \sum_{n=|l|}^{N(l)} F_n^l\ Y_n^l(\phi, \lambda)$$

Expression for the coefficients, F_n^l:

$$F_n^l = \frac{1}{2\pi} \int_0^{2\pi} \int_{-\pi/2}^{\pi/2} F(\phi, \lambda)\ P_n^l(\sin(\phi)) e^{-il\lambda} \cos(\phi)\ d\phi\ d\lambda$$

The idea is to advance the solution in spectral space. However, the nonlinear terms are evaluated in physical space, and this requires a transform from spectral to physical space and back, ***at each time step***.

Transforms for the Spherical Harmonic Expansion

For the evaluation of the integrals in the coefficients, we need the following operations

1. Along latitudes, use the trapezoidal rule and the FFT (exploit periodicity)
2. Along longitudes, use Gaussian quadrature, i. e. weighted sums

The transform has an "East-West" phase, with no data dependencies between different latitudes, followed by a "North-South" phase, with independent work for each longitude.

Thus, there is a great deal of inherent parllelism in the transforms.

Advanced Computer Architectures

- Cray Y-MP, Cray 3 (?)
- Connection Machine (CM2)
- Hypercubes (iPSC/2, iPSC/860, nCUBE)
- Shared Memory (Alliant, Sequent, Encore, ...)
- Mesh Connected, Distributed Memory (Intel Touchstone)

Parallel Performance of a 2D FFT on the Intel iPSC/860

(Coarse/Medium Grained Parallelism)

Require independent 1-D FFT's along grid rows, followed by independent FFT's along grid columns

Each processor handles one or more rows/columns of the computational grid. FFT's are performed locally. Data movement required between "X-Phase" and "Y-Phase"

This problem closely related to the Spherical Harmonic transforms, in terms of the structure of the data dependencies.

Parallel Performance of a 2D FFT on the Intel iPSC/860

Times for a fast Poisson solver based on 2-D FFTs on the iPSC/860.

All times are in milliseconds[1].

P	Grid Size	Total	2d FFT			Vector Divide	Inverse 2d FFT		
			FFT in x-dir	Trans-pose	FFT in y-dir		FFT in y-dir	Trans-pose	FFT in x-dir
64	127 × 127	19.6	1.4	6.2	1.7	1.0	1.7	6.3	1.4
	255 × 255	60.0	4.8	16.8	7.3	1.9	7.3	16.9	5.0
	511 × 511	221.4	20.1	56.5	32.2	3.8	32.2	56.4	20.2
	1023 × 1023	931.8	98.5	212.5	151.0	7.5	151.0	212.8	98.5
128	127 × 127	14.9	0.9	5.4	0.7	0.9	0.7	5.4	1.0
	255 × 255	36.6	2.8	11.2	3.4	1.8	3.4	11.2	2.8
	511 × 511	125.5	10.5	35.0	15.4	3.7	15.4	35.0	10.5
	1023 × 1023	506.1	50.0	125.3	74.0	7.5	74.0	125.3	50.0

[1] S. Cote, Master's thesis, May 1991.

Planned Implementation on the Connection Machine (CM2)

(Fine-grained Parallelism, Hypercube Topology)

Imbed a 2-D processor mesh in the CM2, using the binary-reflected Gray code. This ensures that each row and column in the processor mesh is a sub-cube.

Feasible to assign one grid point to one processor. In contrast to the Intel hypercube, it is more natural to do the FFT's and the Gaussian quadrature in parallel, without a costly data movement operation between the two phases. There is no penalty associated with the SIMD nature of the Connection Machine, due to the regular structure of the transforms.

Planned Implementation on the Connection Machine (CM2)

This planned implementation has the potential to outperform current Global Spectral Models that have been optimized for machines like an eight processor Cray Y-MP. On the full CM2, the larger memory can also lead to better resolution.

Functional Programming Languages

Samuel Kamin

- Comparing languages for "maintainability"
- Functional languages and scientific programming
- Efficiency problems of functional languages
  - Array comprehensions
  - Destructive update

ICLASS review — Oct. 18, 1991

Comparing languages for "maintainability"

To what extent is it possible to share code among multiple "versions" of a program?

Interpreters from book **Programming Languages: An Interpreter-based Approach** are the different versions. Original interpreters written in Pascal ($\approx$ 800–1200 lines each). No code was directly shared.

Question: What language would lead to optimal sharing of code?

Experiment

Programming three versions — evaluators for simple language (`chap1`), `lisp`, and `scheme` — in two languages (C++, Scheme), attempting to achieve optimal "fair" reuse.

Comparing languages for "maintainability"

For both C++ and Scheme versions, code can be divided into five parts: "shared" (by all versions), "chap1", "lisp/scheme" (shared by those two versions), "lisp" and "scheme".

Definitions:

CHAP1 = shared + chap1
LISP = shared + lisp/scheme + lisp
SCHEME = shared + lisp/scheme + scheme
$|P|$ = size of program P (in characters, excluding leading blanks)
Total = shared + chap1 + lisp/scheme + lisp + scheme
Unshared-total = CHAP1 + LISP + SCHEME

	$\frac{\|\text{Total}\|}{\|\text{Unshared-total}\|}$	$\frac{\|\text{shared}\|}{\|\text{Total}\|}$
C++	.46	.45
Scheme	.42	.57

	$\frac{\|\text{shared}\|}{\|\text{CHAP1}\|}$	$\frac{\|\text{shared}\|}{\|\text{LISP}\|}$	$\frac{\|\text{shared + lisp/scheme}\|}{\|\text{SCHEME}\|}$
C++	.84	.63	.79
Scheme	.91	.70	.87

C++ vs. Scheme
A: Parsing

One of the re-use problems is that the main parsing routine, **`parseExpr`** changes with each version. However, it is mutually recursive with functions that do not change.

C++ solution: Class hierarchy for parsing:

Parser
- Chap1_parser
- Lisp_parser
  - Scheme_parser

Scheme solution: Function to create parsers:

`shared`:
```
(define (mkParser isValue parseValue parseApplic printVal)
```
`chap1`:
```
(mkParser isNumber? parseNumber parseApplic print-num)
```
`lisp`:
```
(mkParser isLispValue? parseLispValue parseApplic SExpPrint)
```
`scheme`:
```
(mkParser isLispValue? parseLispValue SchemeParseApplic SExpPrint)
```

C++ vs. Scheme
B: Evaluation

Another re-use problem involves dealing with different types of functions, e.g. arithmetic operations, predicates, user-defined functions. The problem arises in part because of differing representations of values. Thus, we need to coerce functions on integers to functions on values.

C++ solution: Class hierarchy for functions:

`Function`
- `StrictFun` – functions that evaluate arguments
  - `UserFun`
  - `PrimArithFun` – functions whose args and results are values
    - `BinaryFun` – binary PrimArithFun

Then, for example, addition is defined as an instance of `BinaryFun`.

Scheme solution: Functions to coerce other functions:

`(mkStrictFun f)` – f a function from values to values;
– result is function from expressions to values

`(mkBinaryOp f)` – f a function of two values;
– result is function of lists of values

`(mkBinArithFun f)` – f a function of two integers;
– result is function of two values

Then addition is defined as the function:

```
(mkStrictFun (mkBinaryOp (mkBinArithFun +)))
```

Comparing languages for "maintainability"

- Functional programming ideas are useful for "software engineering."
- Objects in object-oriented languages are really just functions by a different name.
- Functional languages provide better structuring capabilities than object-oriented languages, and provide them in a more direct form.

Functional languages and scientific computation

Three advantages:

- General structuring capabilities previously discussed — generalized "frameworks"
- Lazy evaluation allows for modularization of iterative processes
- Convenient "array comprehension" notation

Disadvantage: inefficiency

Lazy evaluation allows for modularity in iterative processes

Sequence of approximations is a first-class value — even though it is infinite. Can define operations on such sequences to improve approximations, without altering original program.

E.g. Iterative calculation

Fortran

```
LAST = X0
THIS = NEXT(THIS)
DO WHILE (THIS-LAST >= EPS)
   LAST = THIS
   THIS = NEXT(THIS)
END WHILE
```

Haskell

```
gen_approx x = x : gen_approx (next x)
approx_seq = gen_approx x0
find p (x1:x2:xs) = if p(x1, x2)
   then x2 else find p (x2:xs)
find_answer = find (\x, y -> x-y < eps)
answer = find_answer approx_seq
```

E.g. Same calculation, with improvement

```
LAST = X0
THIS = NEXT(THIS)
NEWLAST = X0
NEWTHIS = IMPROVE(LAST, THIS)
DO WHILE (THIS-LAST >= EPS)
   LAST = THIS
   THIS = NEXT(THIS)
   NEWLAST = NEWTHIS
   NEWTHIS = IMPROVE(LAST, THIS)
END WHILE
```

```
improves (x1:x2:xs) =
   improve(x1, x2) : improves(x2:xs)
answer = find_answer
       (improves approx_seq)
```

Array comprehension notation

The following is a dynamic programming solution to the problem of computing the probability of winning series of games. Player has already won x games and lost y games in a first-to-reach-n-wins contest. Each game has a 50% chance of being a win or a loss.

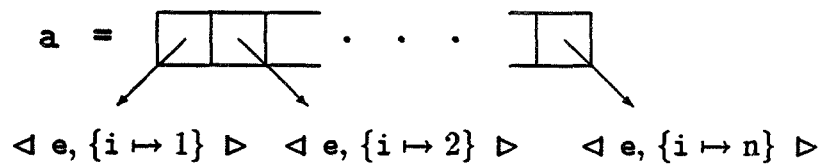
```
main = print (series 8 1 0)

-- series :: Int -> Int -> Int -> Double
--   n == number of games in series
--   x == number of games already won
--   y == number of games already lost
series n x y =
    P!(n-x, n-y)
    where {
     P = array ((0,0), (32,32))
             ( [((0,j) := 1.0) | j <- [1 .. 32]] ++
               [(i,0) := 0.0) | i <- [1 .. 32]] ++
               [(i,j) := 0.5*P!(i,j-1) + 0.5*P!(i-1,j) |
                     i <- [1 .. 32], j <- [1 .. 32]] )
    }
```

Efficiency problems of functional languages

- Strictness analysis
- Array comprehensions: In general, array elements depend arbitrarily on other array elements:

```
a = array [i := ... a!(i-1) ... a!(i/2) ... | i <- [1..n]]
```

Thus, `[i := e | i <- [1..n]]` must create array like this:



Each evaluation `a!i` causes a context switch.

- Destructive update: In lazy languages, cannot use destructive assignment (`A[i] := x`) because sequence of events is too unpredictable. However, this can make simple operations extremely inefficient:

```
iterate_matrix(A)
   = if p(A) then A
      else iterate_matrix(f(A))
   where f(A) = ... update A at some points ...
```

Without major static analysis, this requires creation of a copy of `A` (both modified and unmodified parts) at each iteration.

Compiling array comprehensions efficiently

Our research has centered on producing efficient code for recursive array comprehensions:

$$a = array\ [(f_1(i,j), f_2(i,j)) := e[i,j] | i \leftarrow [i_0..i_1], j \leftarrow [j_0..j_1]]$$
$$++ [(f'_1(i,j), f'_2(i,j)) := e'[i,j] | i \leftarrow [i'_0..i'_1], j \leftarrow [j'_0..j'_1]]$$
$$\vdots$$

where e, e', ..., may have references to a (see function `series` above).

- Transform to recursive form:

$$a = \textit{if } (x, y)=(f_1(i,j), f_2(i,j) \textit{ for some } i, j$$
$$\textit{then } a!(x, y) := e$$
$$\textit{else if } (x, y)=(f'_1(i,j), f'_2(i,j) \textit{ for some } i, j$$
$$\textit{then } a!(x, y) := e'$$
$$\vdots$$

  Problem is inverting functions f_1, f_2, ... For linear functions, have achieved approximate doubling in speed.

- Transform recursive function to iterative form: Have made some progress here, but no reliably efficient solution will be found.

Destructive update

Destructive update can be accommodated if updates can be serialized by the programmer. One way to accomplish this is to use type-checking.

New types:	`Ref` τ	(reference to a τ)
	`Obs` τ	(state-observing value of type τ)
New terms:	`letrec v := e in t`	(declare a reference)
	`get x <= l in t`	(dereferencing)
	`l := e; t`	(assignment)

Then, type-checking rules ensure that expression:

```
f( (l:=e; get x<=l in x), get x<=l in x)
```

is type-incorrect. Legal alternatives are:

```
l:=e; f(get x<=l in x, get x<=l in x)
      l:=e; get x<=l in f(x, x)
get x<=l in (l:=e; get y<=l in f(y,x)),
```

none of which depends upon order of evaluation.

Summary

- Value of functional languages
- Our approach to two efficiency problems:
  - Array comprehension
  - Destructive update
- General problem is space-efficiency and prediction/control of space utilization

Acquiring Software Design Schemas:

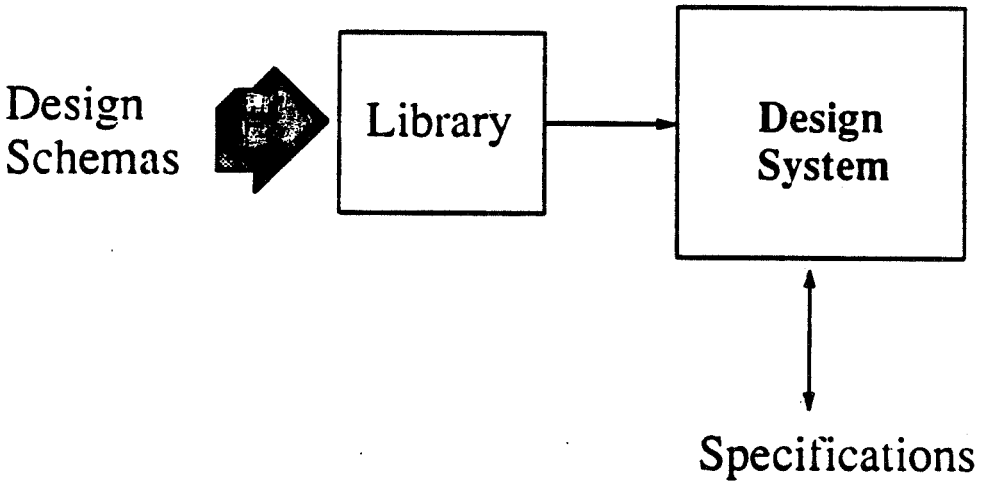
A Machine Learning Perspective

by

Mehdi T. Harandi and Hing-Yan Lee

Dept. of Computer Science
University of Illinois at Urbana-Champaign

Problem



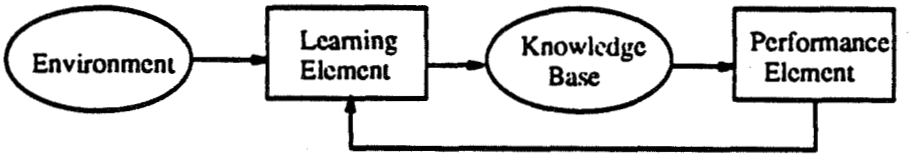
ISSUES

- Can we automate this schema acquisition process?
- To what extent is this automation possible?

Two approaches available:

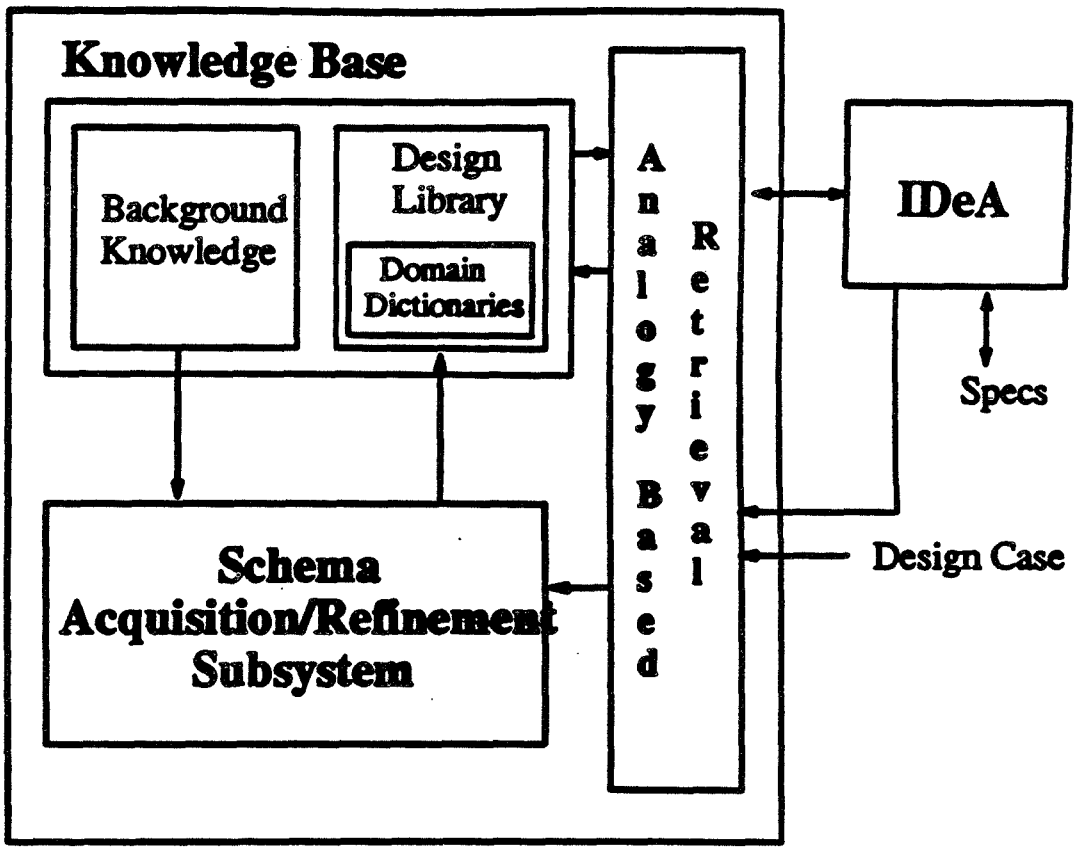
- Model-based
- Case-based

Related Work



- Performance Element = Design Subsystem + User
- Learning Element = Schema Acquisition/Refinement Subsystem
- Environment = Domain Analyst

System Overview



Background Knowledge

```
% datatype.pl
% Data type or background knowledge

basic_type(boolean).
basic_type(character).
basic_type(integer).
basic_type(real).
basic_type(string).
basic_type(monetary_type).
basic_type(temporal_type).

isa(id_type, candidate_key_type).
isa(name_type, candidate_key_type).
isa(title_type, candidate_key_type).
isa(customer_id_type, id_type).
isa(movie_id_type, id_type).
isa(video_id_type, id_type).
isa(tape_id_type, id_type).
isa(dl_id_type, id_type).
isa(timestamp, id_type).
isa(text, string).
isa(title_type, string).
isa(name_type, string).
isa(address_type, string).
isa(tel_no_type, integer).
isa(ind_type, boolean).
isa(date_type, temporal_type).
isa(time_type, temporal_type).
isa(timestamp, temporal_type).
isa(us_currency, monetary_type).
isa(dollar_type, us_currency).
isa(cent_type, us_currency).
isa(dime, cent_type).
isa(nickel, cent_type).
isa(quarter, cent_type).
isa(half_dollar, cent_type).
isa(sterling, monetary_type).
isa(pound, sterling).
isa(penny, sterling).
isa(code_type, string).
isa(code_type, integer).
isa(code_type, character).
isa(tape_format_type, code_type).

% last isa predicate as a catch-all
isa(_, unknown).

is-part-of(time_type, timestamp_type).
is-part-of(date_type, timestamp_type).
```

Top-Level Acquisition Algorithm

Given: a design library of schemas and cases
$L = \{s_1, s_2, \ldots, s_m, c_1, c_2, \ldots, c_n\}$

and a new DesignCase

ACQUIRE (NewDesignCase)

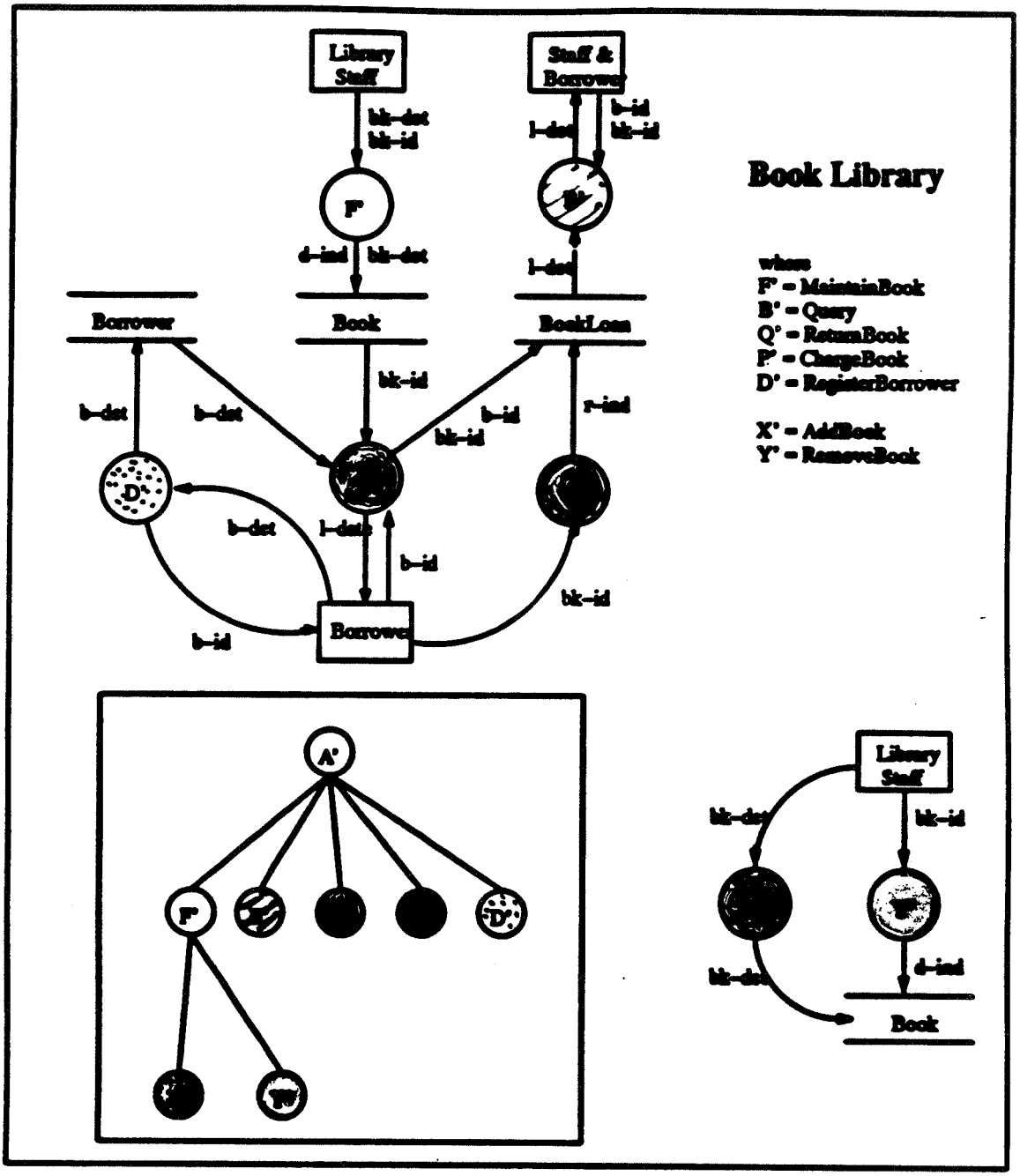
if (RETRIEVE (NewDesignCase, designSchema) $\neq$ FAIL)

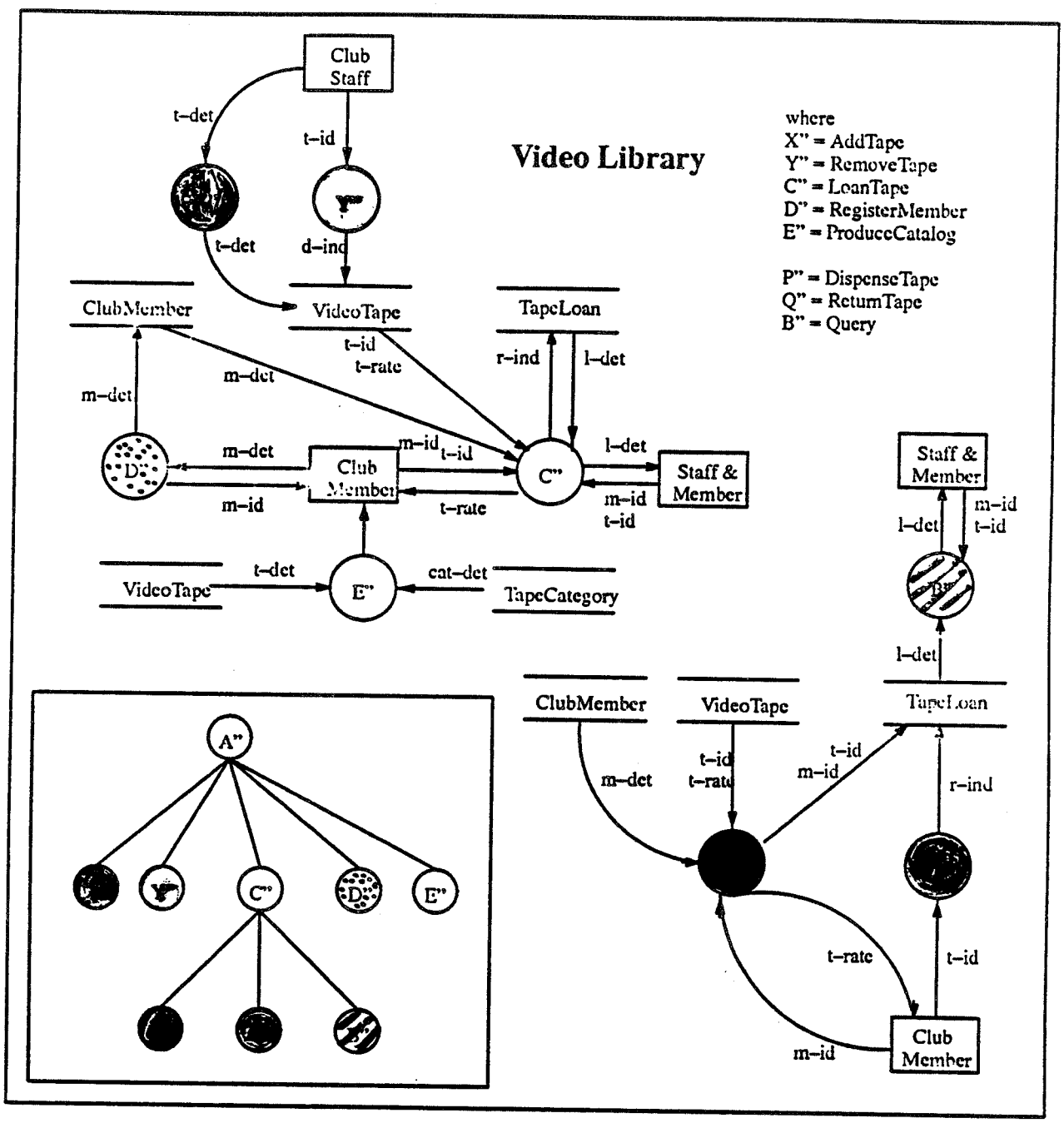
then IMPROVE

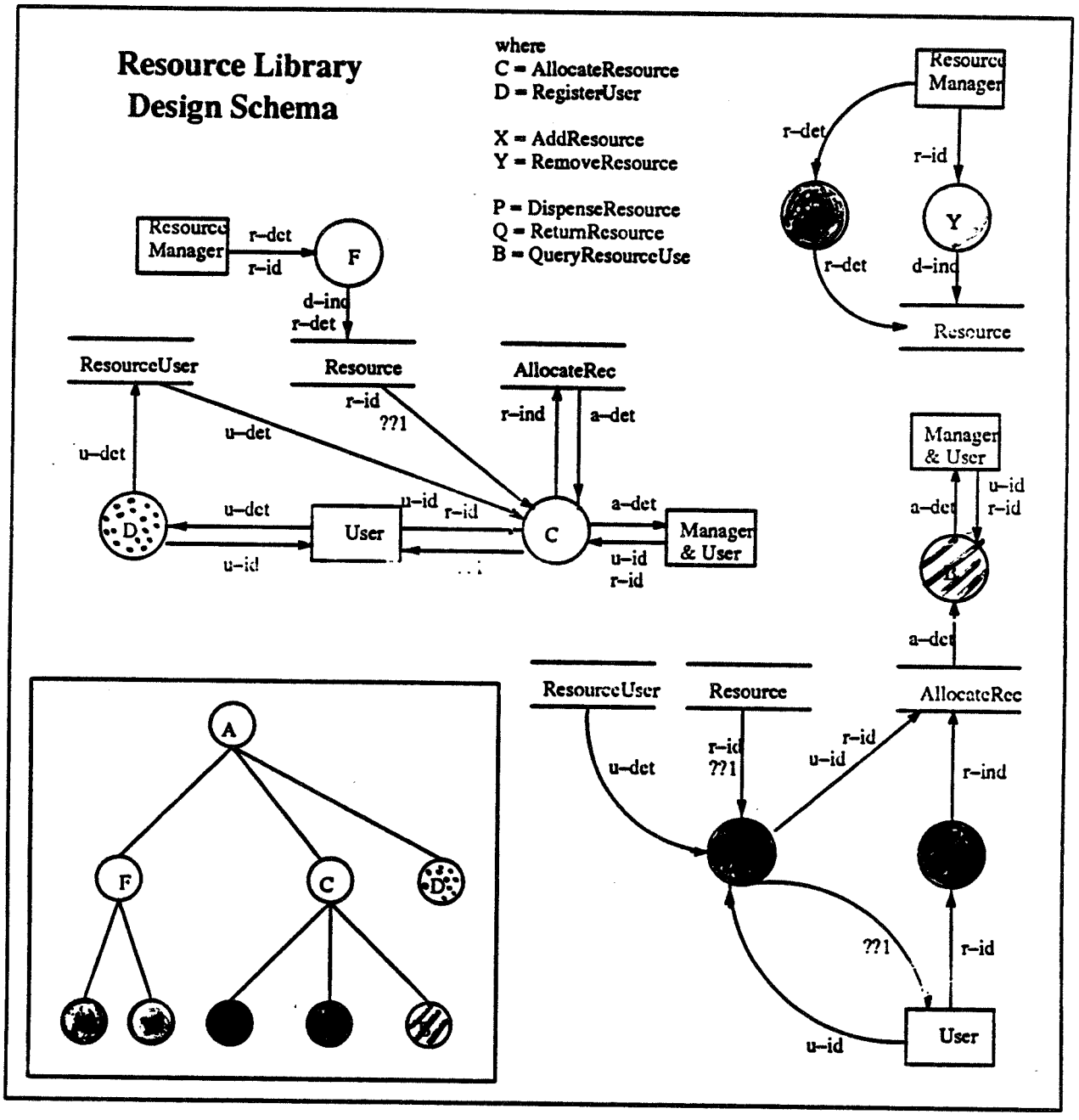
else if (RETRIEVE (NewDesignCase, designCase) $\neq$ FAIL)

then if (threshold exceeded)
then DERIVE-SCHEMA
else STORE (NewDesignCase)

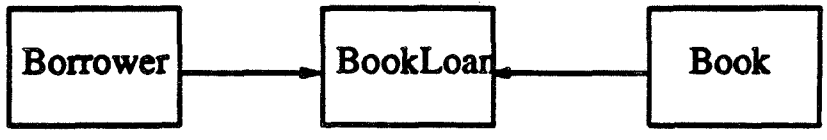
else STORE (NewDesignCase)


Library Staff
Staff & Borrower
bk-dat
bk-id
b-id
l-dat
F'
d-ind
Borrower
Book
BookLoan
b-dat
r-ind
D'
Book Library
where
F' = MaintainBook
B' = Query
Q' = ReturnBook
P' = ChargeBook
D' = RegisterBorrower
X' = AddBook
Y' = RemoveBook
A'
Y'
Library Staff
bk-dat
bk-id
d-ind
Book


Video Library
where
X" = AddTape
Y" = RemoveTape
C" = LoanTape
D" = RegisterMember
E" = ProduceCatalog
P" = DispenseTape
Q" = ReturnTape
B" = Query
Club Staff
t-det
t-id
t-det
d-ind
ClubMember
VideoTape
TapeLoan
t-id
t-rate
r-ind
l-det
m-det
m-det
m-det
m-id
t-id
Club Member
m-id
t-rate
l-det
Staff & Member
m-id
t-id
D"
C"
VideoTape
t-det
E"
cat-det
TapeCategory
Staff & Member
l-det
m-id
t-id
l-det
A"
C"
D"
E"
ClubMember
VideoTape
TapeLoan
m-det
t-id
t-rate
t-id
m-id
r-ind
t-rate
t-id
m-id
Club Member


Resource Library
Design Schema
where
C = AllocateResource
D = RegisterUser
X = AddResource
Y = RemoveResource
P = DispenseResource
Q = ReturnResource
B = QueryResourceUse
Resource Manager
r-det
r-id
F
d-ind
r-det
ResourceUser
Resource
AllocateRec
r-id
??1
r-ind
a-det
u-det
u-det
u-det
u-id
r-id
D
User
C
a-det
Manager & User
u-id
r-id
u-id
Resource Manager
r-det
r-id
Y
r-det
d-ind
Resource
Manager & User
u-id
r-id
a-det
a-det
A
F
C
D
ResourceUser
Resource
AllocateRec
u-det
r-id
??1
r-id
u-id
r-ind
??1
r-id
u-id
User

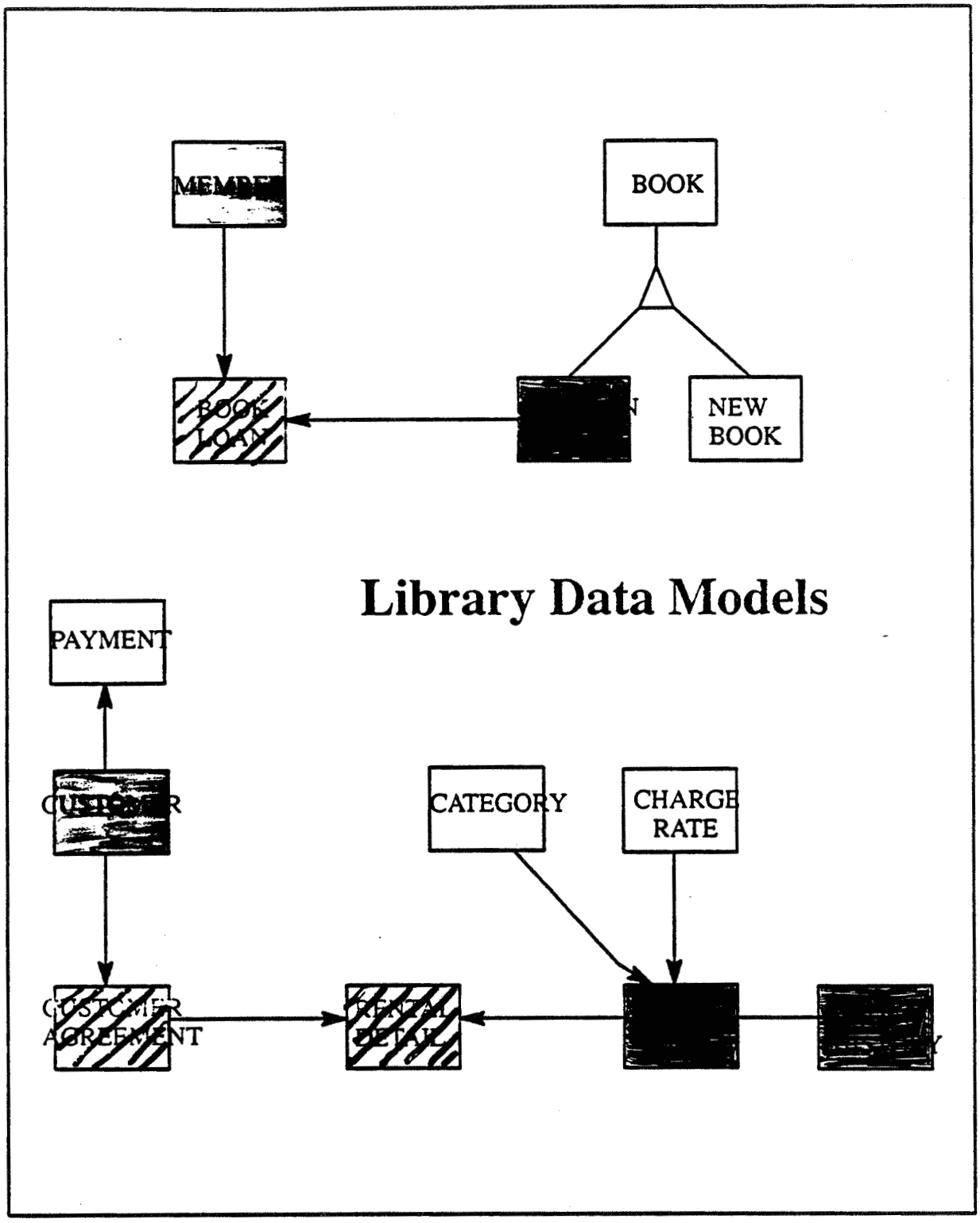
Video Library



Book Library

Resource Library

Data Models


BOOK
NEW
BOOK
Library Data Models
PAYMENT
CATEGORY
CHARGE
RATE

Schema Derivation Mechanism

1. mapping and abstraction of objects and attributes;

2. mapping and abstraction of operations;

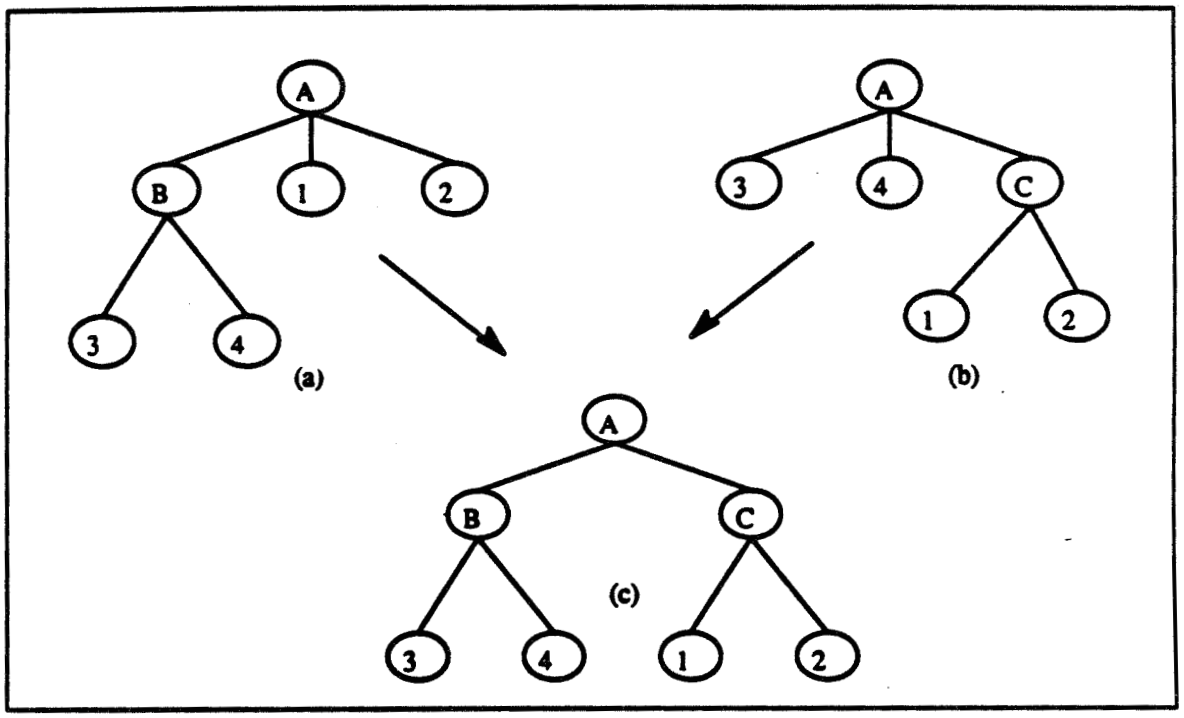
3. validation of design schema;

4. creation of refinement and specialization rules;

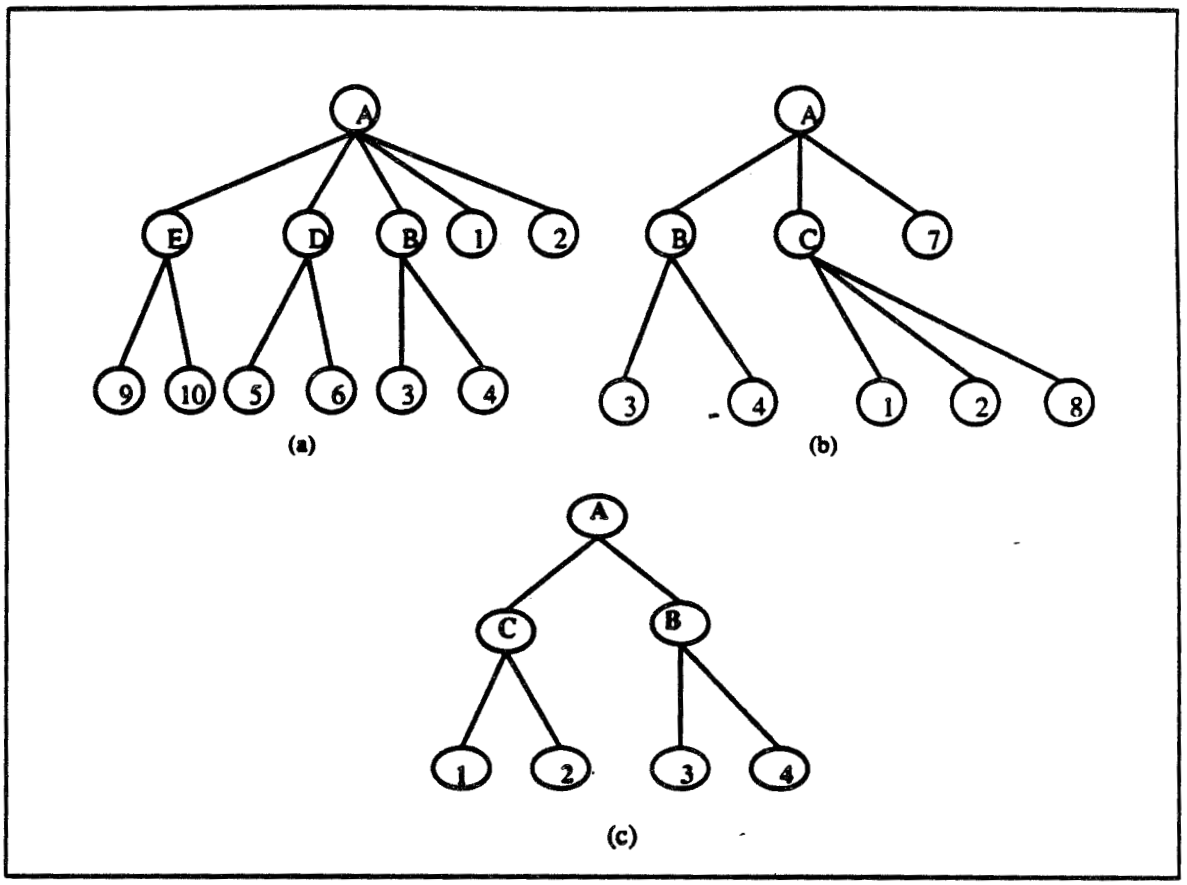
5. cataloging of design schema and rules; and

6. cataloging of object and attribute mappings.

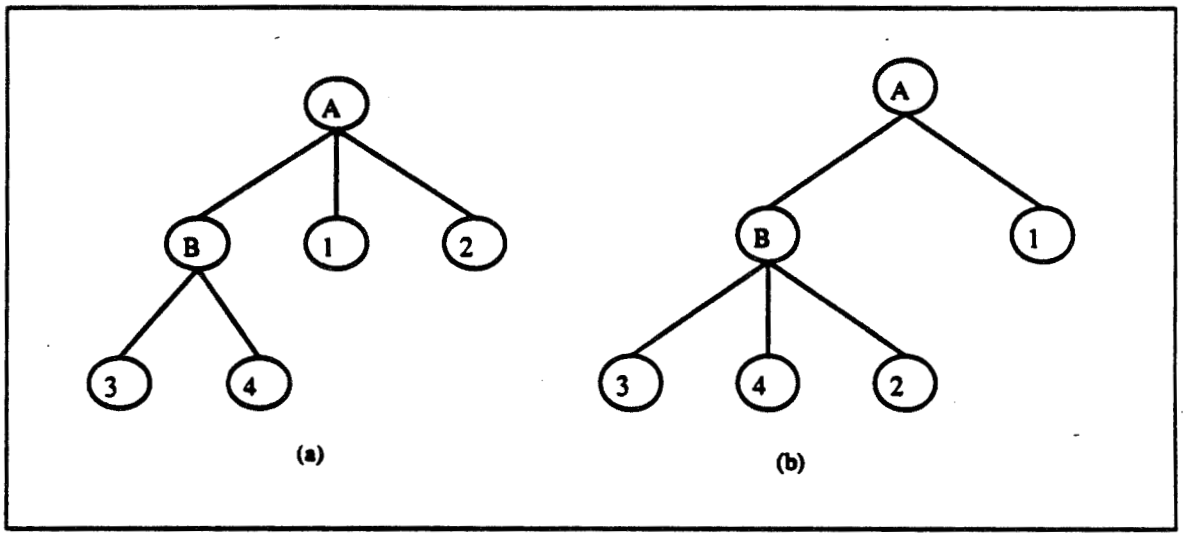
Missing Abstraction Level Discrepancy



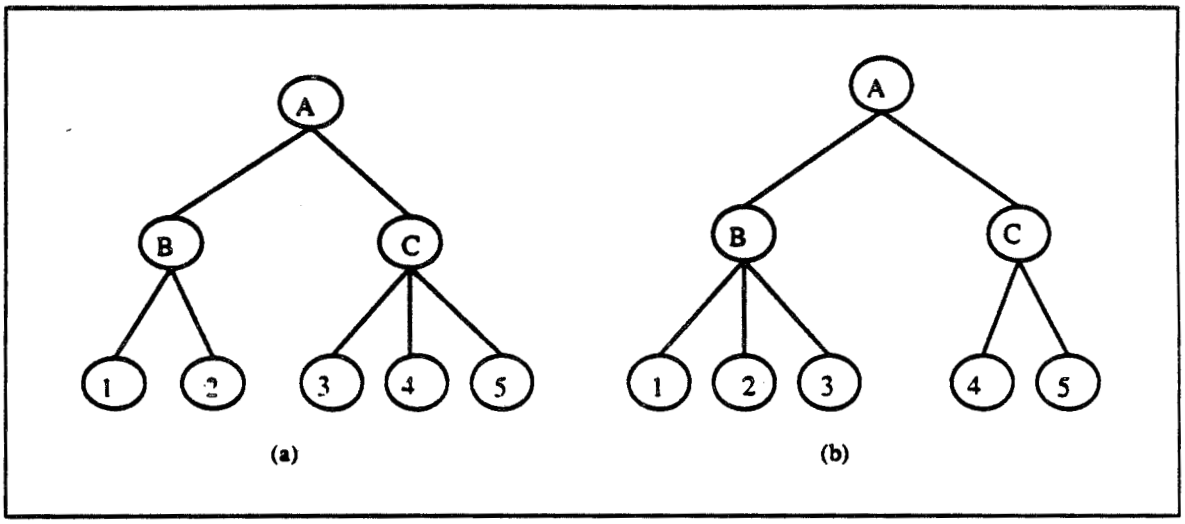
Missing Abstraction Level Discrepancy



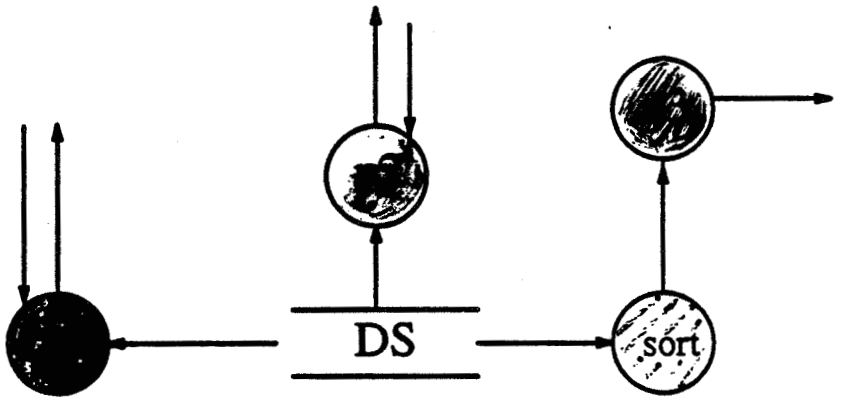
Overlapping Abstraction Level Discrepancy



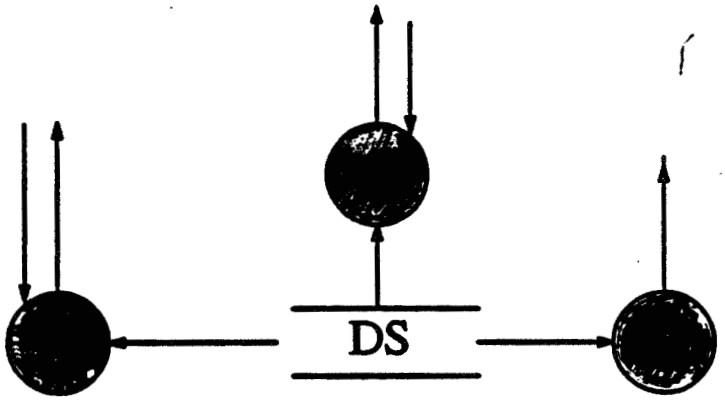
Misplaced Operation Discrepancy



Dataflow Discontinuities



(a)



(b)

p1 = enquiry
p2 = print report1
p3 = print report2

Constructing Abstract DFD Tree

Input Trees Reduction

Delete all unmatched P–ops in the input trees and prune upwards A–ops without P–ops.

Abstract DFD Tree Construction

- Create root of abstract DFD tree. Include A–ops from the input trees in a breadth–first manner by creating an abstract A–op in abstract DFD tree for each A–op in input tree. However, for matching A–ops occurring at different heights, include the corresponding abstract A–op in abstract DFD tree only when the lower of the two matching A–ops is encountered.

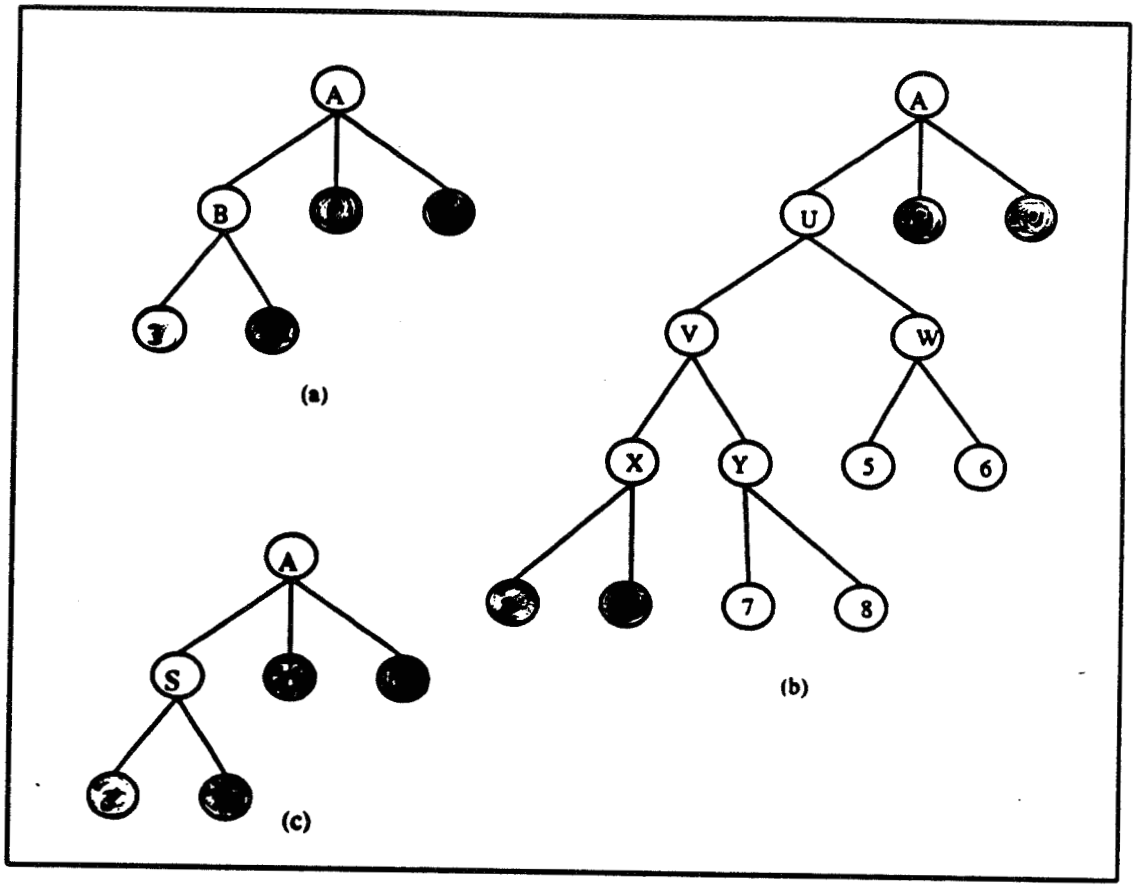
- Include matching abstract P–ops in a bottom–up manner, starting with the lowest–level A–op in abstract DFD tree.

Abstract DFD Tree Pruning

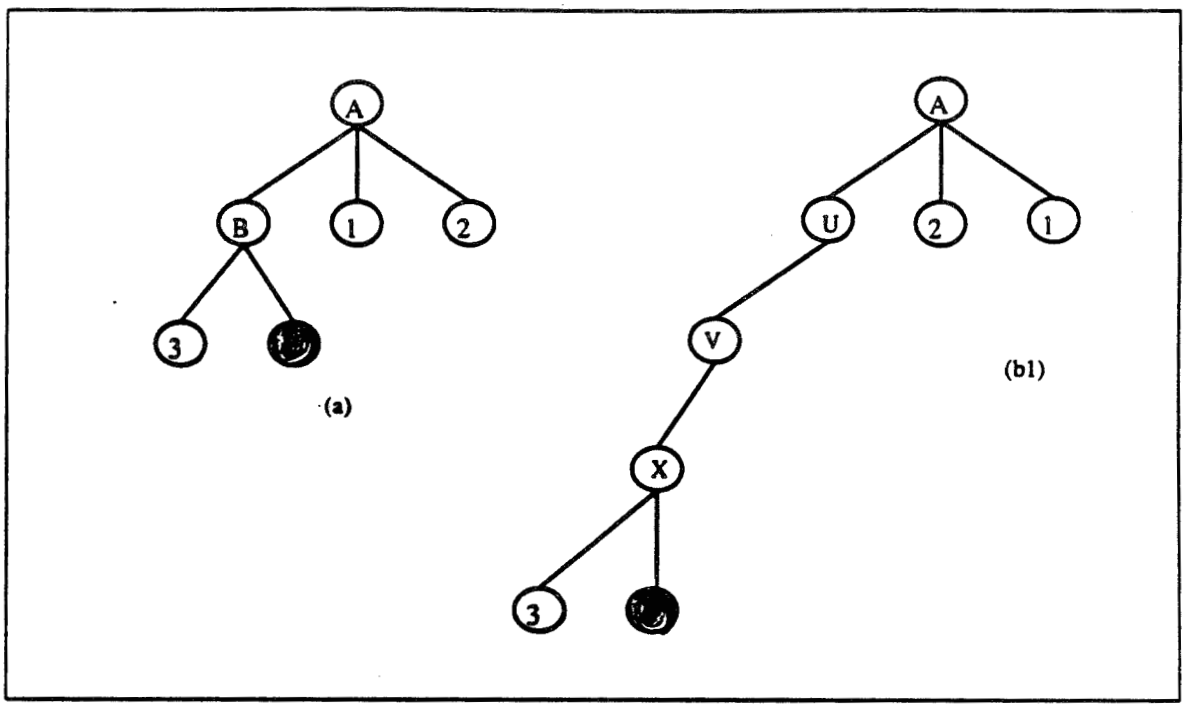
Prune abstract DFD tree to remove unnecessary or spurious abstract A–ops. Delete upwards any abstract A–op with

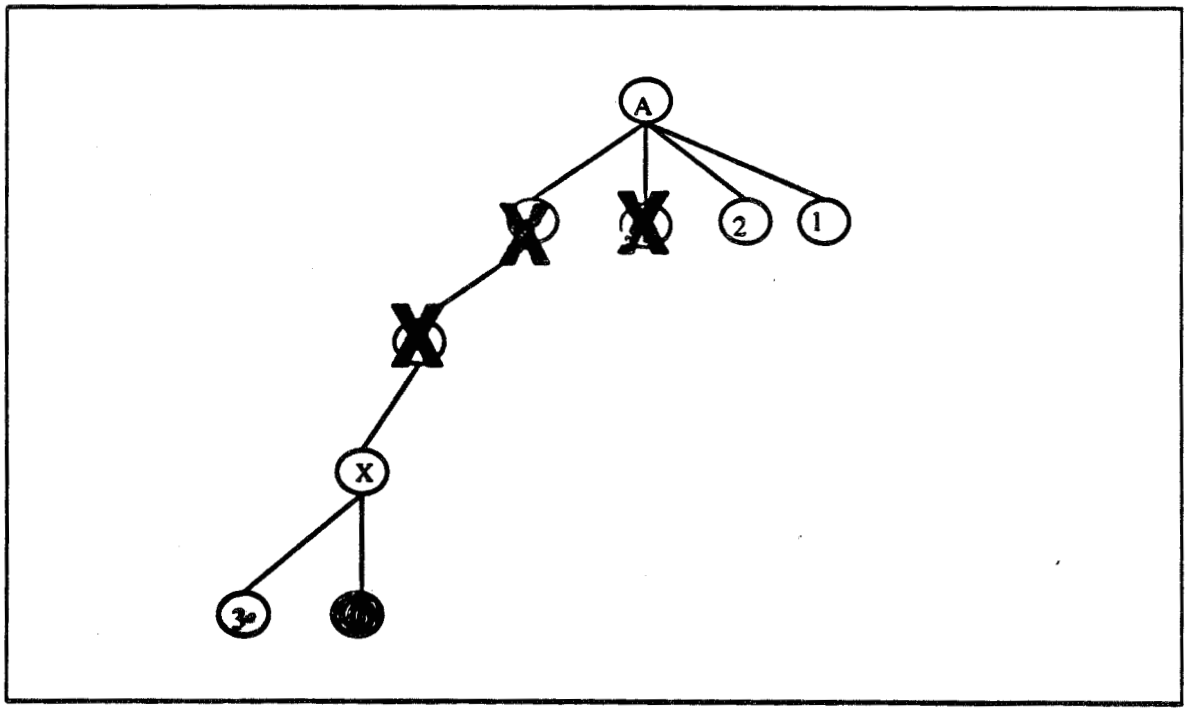
- no or only one abstract P–op; and
- only one abstract A–op.

Example 1

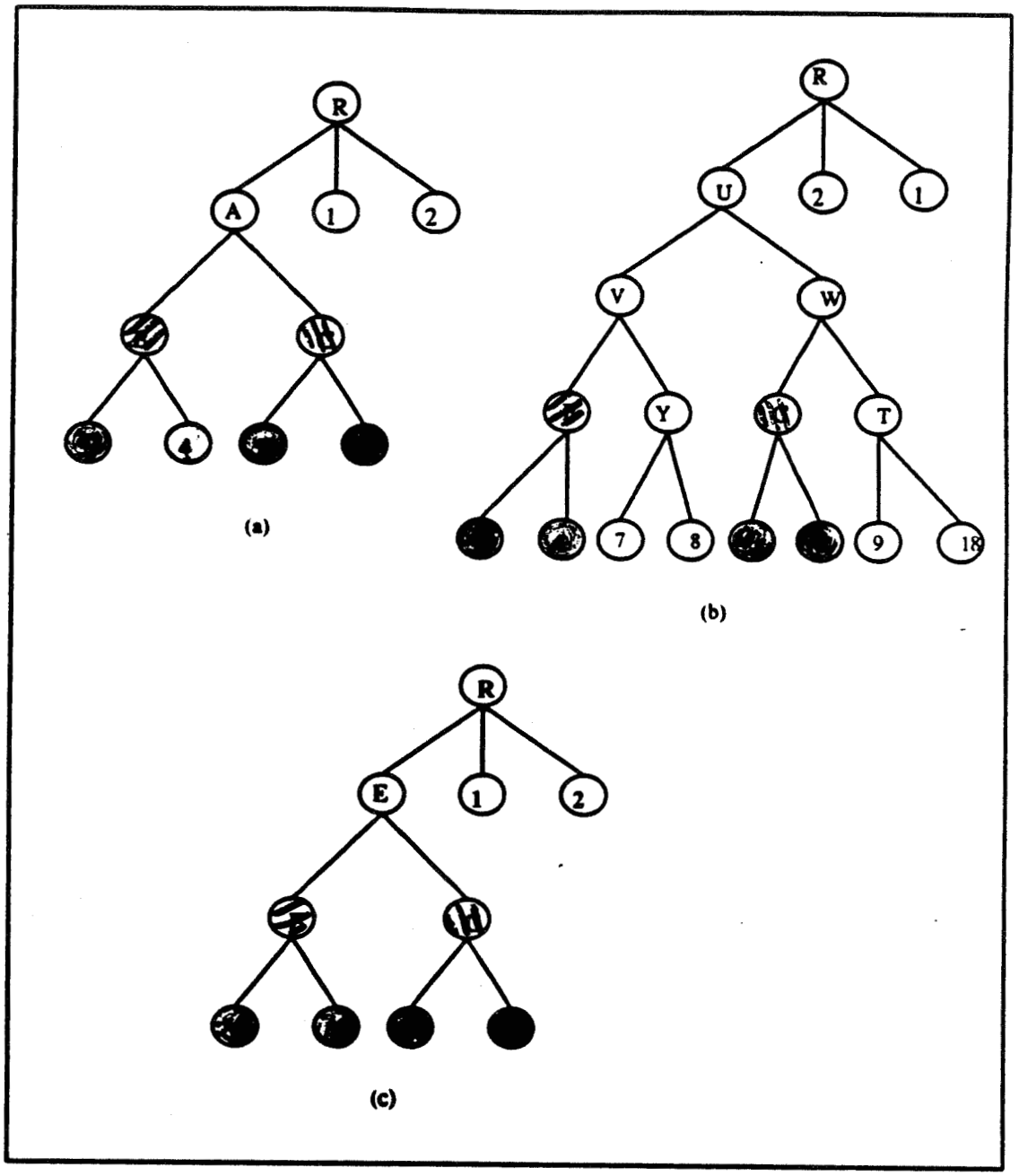


Example 1 (continued)

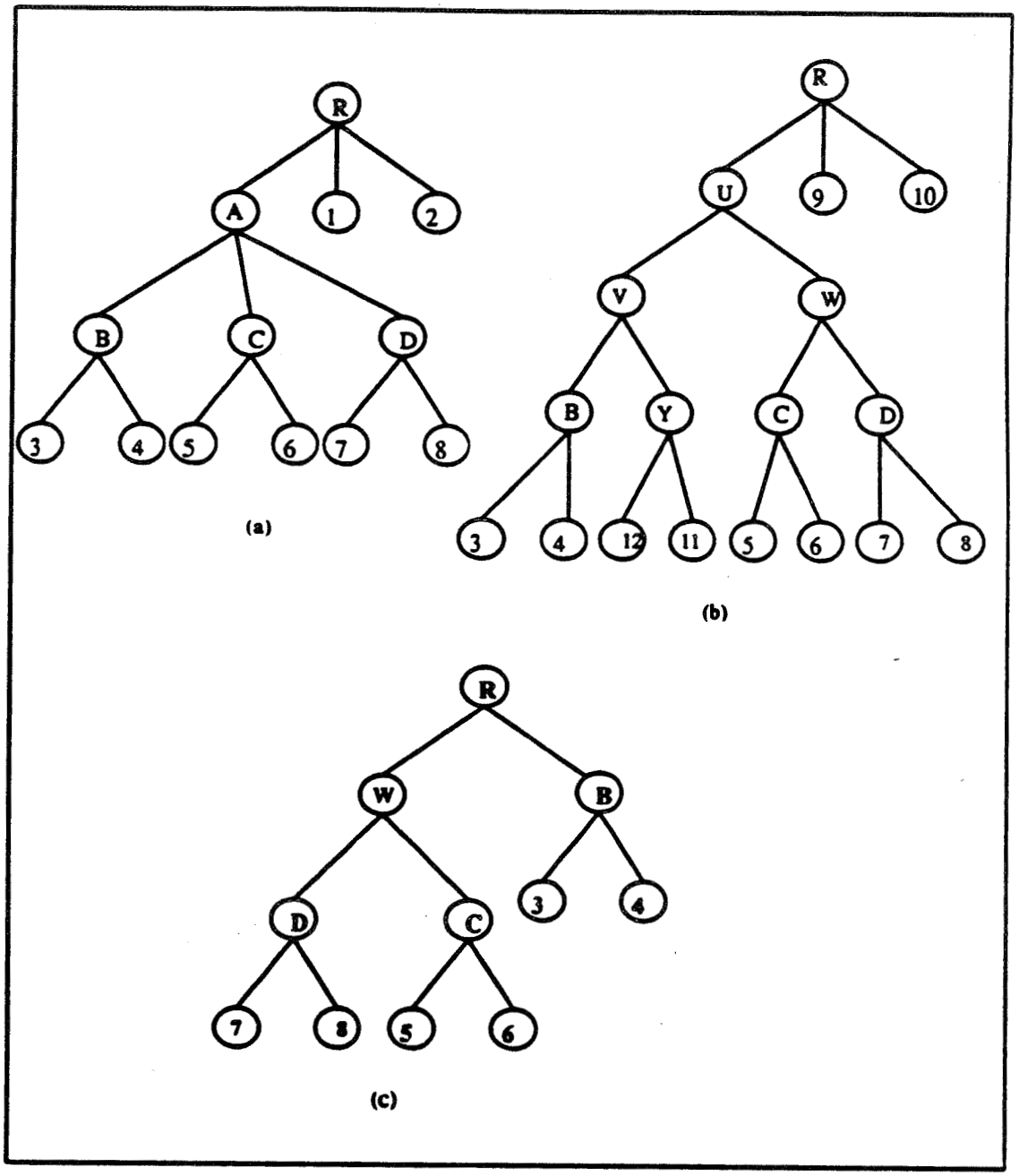




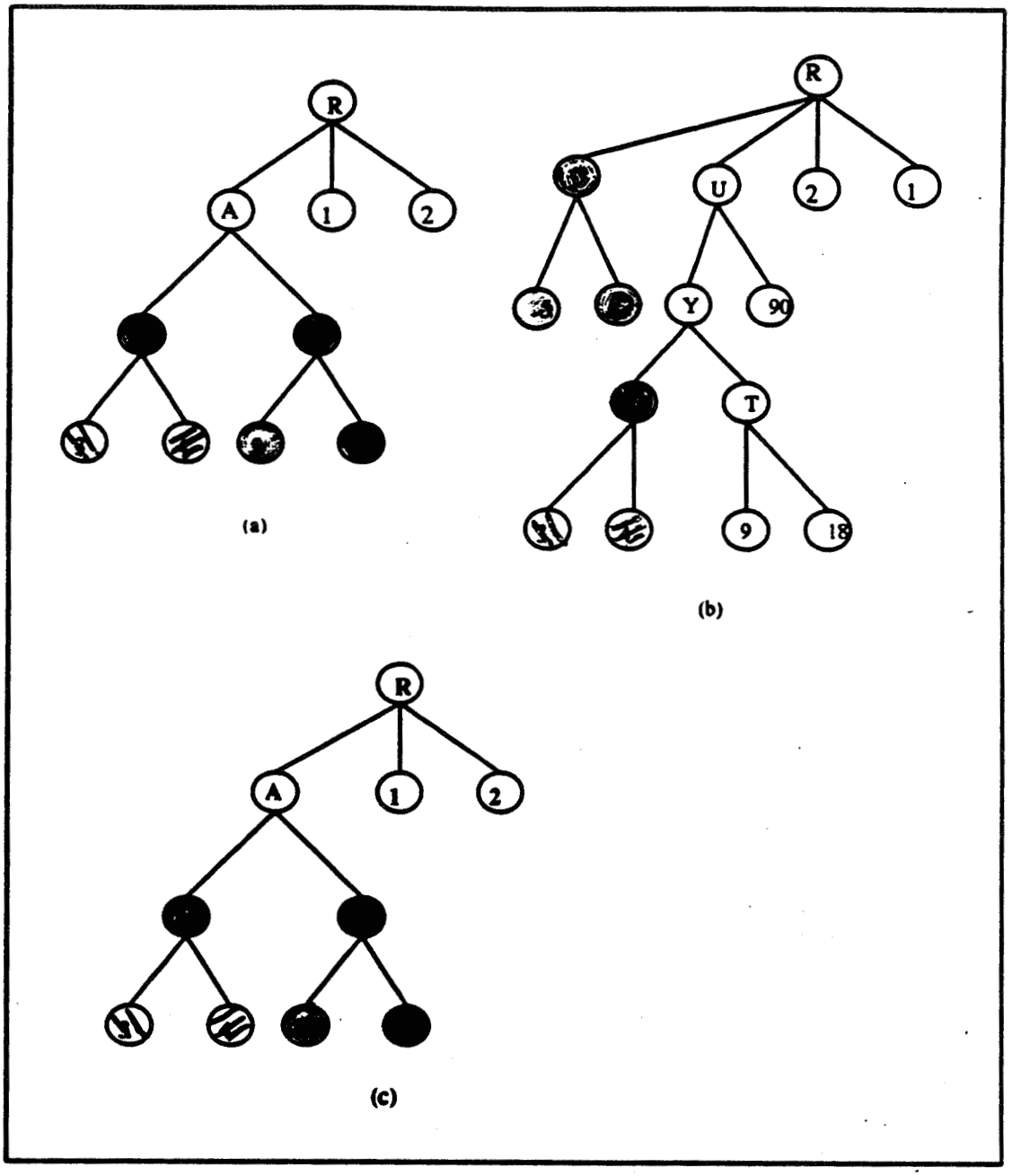
Example 2



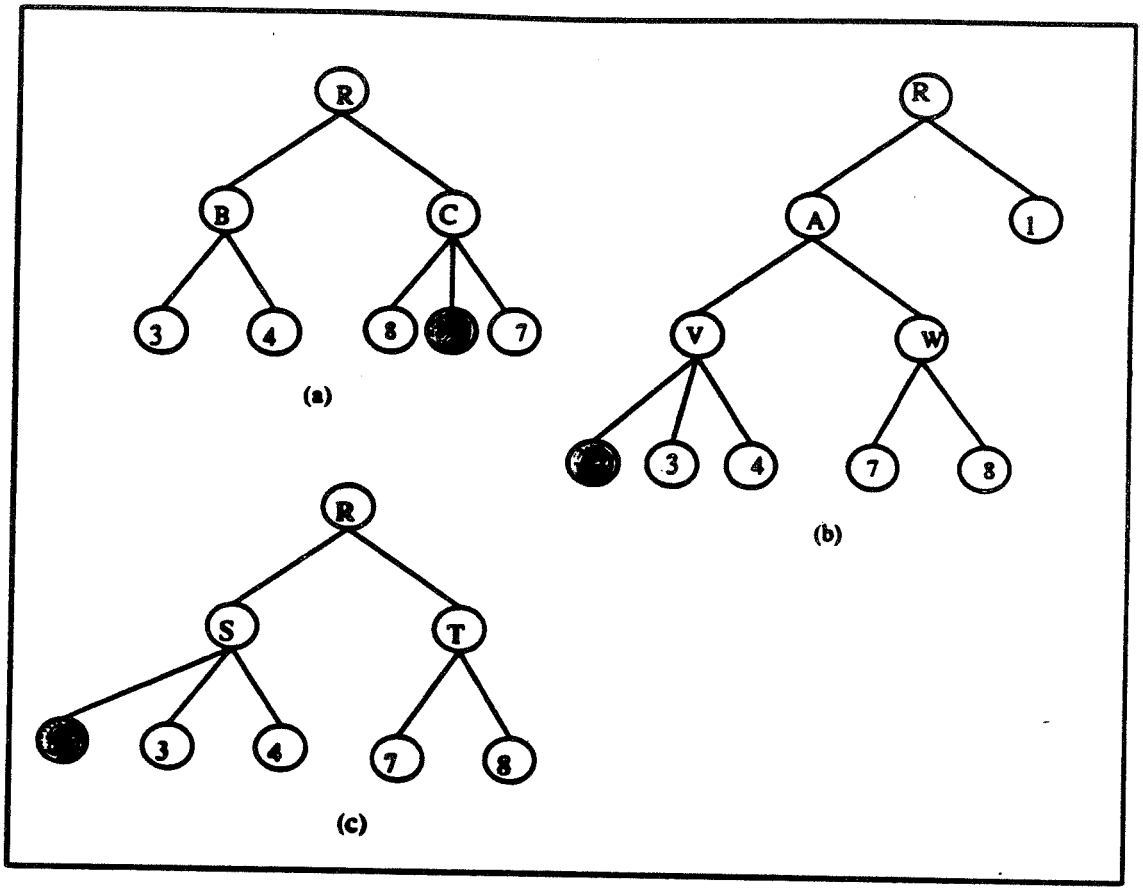
Example 3



Example 4



Example 5



Related Work

- Anderson and Farley (1988)

  - PLANEREUS, a planning system that builds its own hierarchically structured KB.

  - Similarities
    * Creates abstract operators given a set of primitive operators and places it into a taxonomic hierarchy.
    * Creates categories of abstract object types.

  - Differences
    * Levels of abstractions are not present in ABSTRIP-type operators.
    * Relationship links exist between design schemas, but are not used for retrieval.

Reprints

OCTOBER 18, 1991

Reprints on Software Systems, Architectures and Tools; Software Engineering and Artificial Intelligence

(Listed In Order of Appearance)

- Yu, A. C. and K. J. Lin, "A Scheduling Algorithm for Replicated Real-Time Tasks", to appear in *Proceedings of the 11th IEEE International Phoenix Conference on Computers and Communications*, 1992.

- Vrbsky, S. V. and J. W. S. Liu, "An Object-Oriented Query Processor that Produces Monotonically Improving Approximate Answers", in *Proceedings of IEEE International Conference on Data Engineering*, Kobe, Japan, April 1991.

- Liu, J. W. S., K. J. Lin and C. L. Liu, "PERTS: A Prototyping Environment for Real-Time Systems", in *Proceedings of 1991 System Design Synthesis Technology Workshop*, Silver Spring, MD, September 1991.

- Liu, J. W. S., K. J. Lin and C. L. Liu, "Issues in the Imprecise Computation Approach to Fault Tolerance", in *Proceedings of 1991 System Design Synthesis Technology Workshop*, Silver Spring, MD, September 1991.

- Reed, D. A., R. D. Olson, R. A. Aydt, T. M. Madhyastha, T. Birkett, D. W. Jensen, B. A. A. Nazief, and B. K. Totty, "Scalable Performance Environments for Parallel Systems", in *Proceedings of the 6th Distributed Memory Computing Conference*, Portland, Oregon, April 28-May 1, 1991.

- Madany, P. W., R. H. Campbell and P. Kougiouris, "Experiences Building an Object-Oriented System in C++", presented at the TOOLS '91 Workshop, March 4-8, 1991, Paris, France.

- Madany, P. W. and R. H. Campbell, "Organizing and Typing Persistent Objects within an Object-Oriented Framework", to be presented at the Twenty-fifth Hawaii International Conference on System Sciences, Kauai, Hawaii, January 7-10, 1992.

- Campbell, R. H., N. Islam, R. Johnson, P. Kougiouris and P. Madany, "Choices, Frameworks and Refinement", to be presented at the 1991 International Workshop on Object-Orientation in Operating Systems, Palo Alto, CA., October 17-18, 1991.

- Frisch, A. M. and R. B. Scherl, "A General Framework for Modal Deduction", in Allen, J. A., Fikes, R., and E. Sandewall (eds), *Principles of Knowledge Representation and Reasoning: Proceedings of the Second International Conference*, San Mateo, CA: Morgan Kaufmann, 1991.

- Page, C. D. Jr. and A. M. Frisch, "Generalizing Atoms in Constraint Logic", in Allen, J. A., Fikes, R., and E. Sandewall (eds), *Principles of Knowledge Representation and Reasoning:*

Proceedings of the Second International Conference, San Mateo, CA: Morgan Kaufmann, 1991. (Slightly revised version)

- Reddy, U. S. and S. N. Kamin, "On the Power of Abstract Interpretation", submitted for publication.

- Kamin, S. "Head-strictness is not a monotonic abstract property", to appear in *Information Processing Letters.*

- Swarup, V., U. S. Reddy and E. Ireland, "Assignments for Applicative Languages", *Functional Programming Languages and Computer Architecture*, Springer-Verlag.

- Harandi, M. T. and H. Y. Lee, "Acquiring Software Design Schemas: A Machine Learning Perspective", in *Proceedings of the 6th Annual Knowledge-Based Software Engineering Conference*, September 22-25, 1991, Syracuse, NY.

NASA 5)-61
93034
245132
22 p.

A Scheduling Algorithm for Replicated Real–Time Tasks

Albert C. Yu and Kwei–Jay Lin

Department of Computer Science
University of Illinois at Urbana–Champaign
1304 W. Springfield Avenue
Urbana, Illinois 61801

Abstract

We present an algorithm for scheduling real–time periodic tasks on a multiprocessor system under fault–tolerant requirement. Our approach incorporates both the redundancy and masking technique and the imprecise computation model. Since the tasks in hard real–time systems have stringent timing constraints, the redundancy and masking technique is more appropriate than the rollback techniques which usually require extra time for error recovery. The imprecise computation model provides flexible functionality by trading off the quality of the result produced by a task with the amount of processing time required to produce it. It therefore permits the performance of a real–time system to degrade gracefully. We evaluate the algorithm by stochastic analysis and Monte Carlo simulations. The results show that the algorithm is resilient under hardware failures.

Keywords: real–time systems, imprecise computation, software redundancy, periodic jobs, linear programming, multiprocessor scheduling, fault tolerance.

This research was supported in part by the National Aeronautics and Space Administration (NASA) Graduate Student Researchers Program, by NASA under grant NASA–NAG–1–613, and by the Office of Naval Research under grant N00014–89–J–1181.

To appear in the proceedings of the 11th IEEE International Phoenix Conference on Computers and Communications, 1992.

1. Introduction

Many computer applications require fault–tolerant computations with ready–time and deadline constraints [1, 11, 13, 21, 24]. Examples include on–line medical databases, embedded monitors for avionic coordination, intelligent vehicle systems, life supporting systems, and nuclear power plants. It is critical for these embedded real–time systems to complete the computations on time. Otherwise, a timing failure is said to occur and the consequence may be monetary loss or life threatening.

Hard real–time systems, by definition, must have predictable behavior and must be fault–tolerant. If hard real–time systems cannot tolerate certain hardware or software failures, the computations may be interrupted and delayed, resulting in timing failures. Two common approaches to tolerating computer component failures are replication and masking, and rollback and retry [2]. For time–critical systems such as hard real–time systems, the replication and masking approach is more appropriate than the rollback and retry approach which usually requires extra time for fault recovery.

The *imprecise computation* model can be used for managing the loss of computational resources due to hardware failures [5, 16]. The imprecise computation model trades off the quality of the result produced by a task with the amount of processing time needed to produce it. Despite the loss of some computational resources, the imprecise computation model can produce lower quality yet acceptable results with less processing time. In a computer application that supports the imprecise computation model, a task can be logically decomposed into a hard task and a soft task. The hard task must be completed before its deadline. The soft task is an optional task which may not be executed to completion, if insufficient processing time is available. The imprecise computation model provides hard real–time systems with flexible functionality and permits their performance to degrade gracefully in case of hardware failures.

In this paper, we present the *allocate* algorithm for solving the replicated imprecise task allocation problem [1, 19, 21, 22]. The task system of the problem has n periodic tasks each of which can be logically decomposed into a hard task and a soft task. Each periodic task may be replicated. The copies or clones of a task must be assigned to distinct processing elements (*PE*'s). The clones allocated to a PE are scheduled by the priority–driven scheduling algorithms (*PDSA*), i.e. the rate monotonic algorithm (*RM*) or the earliest deadline first algorithm (*EDF*) [17]. The processor utilization of a PE is the sum of the utilizations (processing time / period) of the clones allocated to the PE. The schedulability threshold is the maximum processor utilization of a PE below which the scheduling algorithm can guarantee all deadlines of the periodic tasks allocated

to the PE. The threshold is determined by the algorithm used. When the algorithm is RM, the threshold of a PE is $c(2^{1/c}-1)$ where c is the number of clones allocated to the PE [17]. When the algorithm is EDF, the threshold of every PE is one. The objective of our problem is to find for each periodic task (a) the amount of its processing time and (b) the assignment of its clones to the PE's so that the processor utilization of every PE is not more than its schedulability threshold and the total weighted utilization or *TWU* of the task system is maximized. The optimization problem can also be formulated as minimizing the total weighted unscheduled portions of the soft tasks in a task allocation [20, 25].

The allocate algorithm solves the replicated imprecise task allocation problem in three phases. In the first phase, a greedy algorithm computes the upper bound of *TWU* and the upper bound utilization of each task. The greedy algorithm derives these upper bounds by relaxing some constraints in the original problem. In the second phase, two heuristic algorithms are executed with the upper bound task utilizations to generate two different task allocations on the PE's. We then choose the one which is closer to the optimal task allocation. Because the upper bound task utilizations are used, it is likely that the processor utilizations of some PE's exceed the schedulability threshold. It is necessary in the third phase to adjust the utilization of each task so that every processor utilization is not more than its threshold and *TWU* is maximized. We formulate the problems as linear programs (*LP*) and solve them using known efficient LP algorithms.

The rest of the paper is organized as follows. Section 2 briefly describes the related research. We define the system and failure model assumed in our approach and the replicated imprecise task allocation problem in Section 3. Section 4 describes the allocate algorithm. Section 5 presents the stochastic analysis and experimental evaluation of the algorithm. We summarize and discuss the significance of our results in Section 6.

2. Related Research

There are two basic approaches to scheduling periodic tasks on a multiprocessor system. The *partitioning* method divides the task set into groups and assigns each group to a distinct PE. The group of tasks assigned to a PE can then be scheduled by the techniques for a single PE [17]. The other approach is the *non-partitioning* method. It treats the entire collection of PE's as a large virtual PE. The periodic tasks are scheduled dynamically on all PE's, when the requests for executing them arrive.

The non–partitioning method may lead to very low processor utilizations for some task sets when they are scheduled on multiple PE's using RM [6, 18]. Kim and Welch use a non–partitioning load balancing scheme for the applications where rollback–and–retry is an acceptable mode of recovery. They show that the recovery blocks can be executed in a distributed manner with low overheads [13]. The distributed execution of recovery blocks is capable of masking both hardware and software failures. When a hardware failure occurs during the execution of a task, the recovery block or alternate version of the task is retried later and executed on a different PE. When a software failure is uncovered in a task, the recovery block of the task is executed on any non–faulty PE. If successful, the result produced by the recovery block replaces the erroneous result produced by the primary version as the output of the task.

An extremely stringent reliability would require a large number of (primary) clones which may swamp the system and cause it to have long response times. Krishna and Shin suggest to change some primary clones to ghost clones which are backup copies lying dormant until they are activated to take the place of corresponding primary clones or previously activated ghost clones whose processors have failed [14]. They present an algorithm that ensures that backup schedules can be efficiently embedded within the primary schedule to ensure that hard deadlines continue to be met. Both primary and ghost clones are assigned processing time in the primary schedule. Although the ghost clones are initially passive and do not in that state require any processing time, they represent a latent demand for processor time that must be allowed for. For this reason, they affect the scheduling of the primary clones, and tend to lower the efficiency of the primary schedules. Moreover, the algorithm calls an optimal task allocation algorithm and an optimal task scheduling algorithm which they assume are given.

Ramamritham considers a very general multiprocessor task scheduling problem with fault tolerance, communication network, memory, and other resource constraints [19]. A partitioning heuristic rule is designed for allocating periodic tasks to PE's. The heuristic rule groups two communicating tasks together and assigns them to the same PE, if the amount of communication between them is above a tunable threshold and they are not replicas of the same task. Once the allocation of tasks is determined, the multiprocessor scheduling problem is simplified to that of a single PE. Again, several heuristic rules are designed to speed up the search for a feasible schedule.

Chung et al. use the partitioning algorithms designed by Dhall and Liu for allocating imprecise periodic tasks to PE's [5]. They propose several optimal and heuristic algorithms for scheduling imprecise periodic tasks on a single PE [20]. However, Dhall and Liu's algorithms do

not ensure that the clones of a task are assigned to distinct PE's.

The partitioning approach taken by Bannister and Trivedi is the closest to our approach [3]. They design an approximation algorithm for allocating replicated periodic tasks to distinct PE's so that the processor loads of all PE's are balanced. Their analysis shows that the variations in the processor utilizations of any task allocation produced by the algorithm is bounded from above by a small number. In other words, the algorithm is a good load balancing algorithm. We extend their task system to a more general task system. Our algorithm also uses their task allocation algorithm as a subroutine. Hence, their results are derivable from our model, although our objective (maximize *TWU*) is different from theirs (load balancing).

Yu and Lin generalize the non–partitioning method and design an algorithm for scheduling parallelizable imprecise tasks with multiprocessing overheads [25]. A task is parallelizable, if it can be decomposed to a number of concurrent subtasks (clones). The time allocation part of the problem is formulated as an LP and solved using an efficient LP algorithm. To construct a multiprocessor schedule, we also need to decide which task to run on which processor. The task allocation part of the problem is formulated as the subset sum problem and the bipartite matching problem which are then solved using known efficient algorithms. However, the allocation algorithm may assign the clones (concurrent subtasks) of a task to the same PE. We are currently improving the algorithm so that the clones of a task are assigned to distinct PE's. This work and the work presented in this paper show a contrast between the partitioning approach and the non–partitioning approach. The partitioning approach may have lower processor utilizations because of the sub–optimal partitioning of periodic tasks, while the non–partitioning approach has the multiprocessing overheads associated with the migration of concurrent subtasks.

3. Replicated Imprecise Task Allocation Problem

Before defining the replicated imprecise task allocation problem, we describe two schemes for tolerating different types of hardware failures by replication. The real–time application software is assumed to be error–free. We describe the computer system model and the fault–tolerant schemes next.

For our study, we assume the real–time computer is compose of m identical PE's connected by an interconnection network (Figure 1). Each PE has a processor and a local memory. The interconnection network supports multiple physical communication channels between any two PE's. A hardware component failure will cause one or more PE's to malfunction. Let (*operation*) *mode* i characterize the state in which the real–time computer has i non–faulty PE's. So

our system has m operation modes. We assume that the operating system has the capability to detect, isolate, and remove these failed PE's from service [4, 7]. In addition, the repair time is much smaller than the mean time between successive component failures [14]. When the allocate algorithm is applied at run time, it will consume computational resources. The execution of the algorithm is modeled as an aperiodic task and scheduled using the methods proposed in [22].

Using our model, two replication resolution schemes can be used. Suppose there are two periodic tasks T^1 and T^2. In this paper, we use the superscript of a variable to denote the task ID, and the subscript the operation mode. The result produced by T^1 is consumed by T^2. When T^1 and T^2 are replicated, the result produced by each clone of T^1 is broadcast to all clones of T^2. In the first scheme, a clone of T^2 may use any one of the results that arrive on time. The choice is left to each clone of T^2. Hence, neither agreement nor synchronization among the clones of T^2 is needed. If T^1 has c clones, then T^1 can tolerate up to $c-1$ crash/omission/timing failures [4].

The second scheme is similar to the first scheme except each clone of T^2 determines its input by a majority vote of the results that arrive on time. Again, neither agreement nor synchronization among the clones of T^2 is necessary. The additional processing time needed for voting is added to the processing time of T^2. If T^1 has c clones and faulty components may produce identical errors, then T^1 can tolerate up to $\lfloor (c-1)/2 \rfloor$ response/arbitrary failures. On the other hand, if faulty components do not produce identical errors, then T^1 can sustain up to $c-2$ response/arbitrary failures.

The reliability requirement and the nature of a real-time fault-tolerant application determine which resolution scheme to use for each task and specifies the number of clones needed for each operation mode. The second scheme does not handle the case in which a clone of T^2 needs the majority result at the deadline of T^1, since some extra time is needed for voting. This problem is not addressed in this paper. One possible approach is to introduce an earlier deadline at

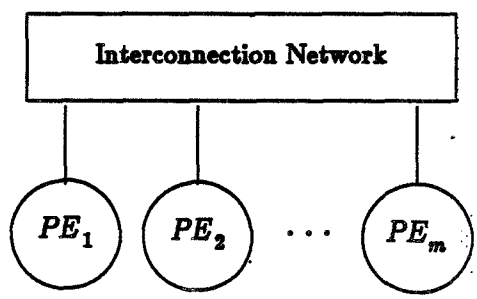


Figure 1: Logical organization of the multiprocessor.

which the clones of T^1 after the data processing can synchronize with one another for voting. The schedulability bounds for periodic tasks with arbitrary deadlines has been studies in the paper [15].

3.1. Problem Definition

A periodic task system, $PTS = \{T^1, T^2, \ldots, T^n\}$, is composed of a collection of periodic tasks. Each task T^j in PTS is characterized by the following rational parameters:

(1) initial request time r^j, the first time instance at which T^j is requested for execution,

(2) period p^j, the duration between any two consecutive requests for executing T^j,

(3) complete processing time τ^j, the duration required to service the request of T^j, and

(4) weight w^j indicating the relative importance of T^j.

A request for executing T^j always arrives at the beginning of a period. The deadline for completing the request is the end of the corresponding period.

Each task T^j in PTS is logically decomposed into a hard task H^j and a soft task S^j. Let h^j and s^j be the processing times of H^j and S^j, respectively, and hence $h^j + s^j = \tau^j$. H^j and S^j are dependent; the execution of S^j cannot begin until H^j is executed to completion. Each task T^j is replicated and has c_i^j identical copies for operation mode i. The number of clones of T^j for mode i is no more than the number of non–faulty PE's (i.e., $c_i^j \leq i$). c_i^j is usually equal to or greater than c_{i-1}^j; thus $< c_m^j, c_{m-1}^j, \ldots, c_1^j >$ is a non–increasing sequence. We use T_i^j to represent the clones $\{C_{i1}^j, C_{i2}^j, \ldots, C_{ic_i^j}^j\}$ of task T^j, and PTS_i the tasks $\{T_i^1, T_i^2, \ldots, T_i^n\}$ for mode i. Hence, PTS_i is a multiset.

The replicated imprecise task allocation problem is consisted of m task allocation problems, one for each operation mode. They are referred as the sub–problems of the replicated imprecise task allocation problem. A task allocation for the sub–problem corresponding to mode i is an assignment of the clones in PTS_i to i identical non–faulty PE's. A task allocation is *feasible*, if

(1) every clone can produce an *acceptable* result before its deadline,

(2) no two clones of a task are assigned to the same PE,

(3) every clone is executed by exactly one PE at each mode, namely no task migration from one PE to another is allowed, and

(4) the processor utilization of every PE is not more than the schedulability threshold.

A clone is said to produce an acceptable result when its hard task is completed before its deadline. The allocate algorithm assigns an equal amount of the processing time (x_i^j) to each clone of task T_i^j. x_i^j should be no less than the processing time (h^j) of its hard task, but is no more than its total processing time (τ^j). The utilization u_i^j of each clone of T_i^j (or the utilization of T_i^j) is x_i^j/p^j.

A feasible task allocation for mode i is *optimal*, if its total weighted utilization $TWU_i = \sum_{j=1}^{n} w^j u_i^j$ is maximum among all feasible task allocations. Since $i \geq 1$, $w^j > 0$, and $u_i^j \geq 0$, TWU_i of feasible task allocations is greater than 0. We can assume that TWU_i of infeasible task allocations is 0.

When either s^j or c_i^j of each task T_i^j in PTS_i is zero, we can show that this sub–problem is NP–complete by transforming the NP–complete bin–packing problem to it [9]. This sub–problem is clearly in NP, since a non–deterministic algorithm can guess a task allocation of the hard clones and verify easily the feasibility of the task allocation. The bin–packing decision problem has a finite set I of items, a size $s(j) \in Z^+$ for each $j \in I$, a positive bin capacity B, and a positive integer i. The problem is to find a partition of I into disjoint sets $I_1, I_2, \ldots, I_i$ such that the sum of sizes of the items in each set is B or less. To transform it to a sub–problem of the replicated imprecise task allocation problem, we let $c_i^j = 1$, $h^j/p^j = \tau^j/p^j = s(j)$, the schedulability threshold be B, and the number of non–faulty PE's be i. The bin–packing problem is thus reduced to an instance of the decision sub–problem corresponding to mode i. Hence, the sub–problem is NP–complete. The replicated imprecise task allocation is at least NP–hard.

4. Allocate Algorithm

It is unlikely that there exists an optimal polynomial time algorithm for solving the replicated imprecise task allocation problem. We propose the heuristic allocate algorithm shown in Figure 2 for solving the sub–problem corresponding to operation mode i. The algorithm has three phases.

Phase 1: Greedy Algorithm

Although it is difficult to solve the sub–problem optimally, an upper bound for the total weighted utilization TWU_i^{upper} and an upper bound utilization u_i^j of each task T_i^j in PTS_i can be computed easily when the scheduling algorithm is EDF. Indeed, if we relax the second and third requirements for feasible task allocations, i.e., if the clones of a task can be on the same PE or migrate, the upper bounds can be computed by a greedy algorithm. When the scheduling

```
Algorithm Allocate
input: PTS, i, PDSA;
output: TWU_i;
begin
  ({u_i^j}, TWU_i^upper) = greedy algorithm(PTS, i, PDSA);    <Phase 1>
  if ((PDSA is EDF) and (u_i^j < h^j/p^j for some j)) then
     Call exception handler;     <No feasible task allocation.>
  else
  begin   <Phase 2>
     A_1 = LUF({u_i^j}, PDSA);    <A_i's are task allocations.>
     A_2 = MF({u_i^j}, PDSA);
     if ((PDSA is EDF) and ((max-proc-util(A_1) ≤ 1) or (max-proc-util(A_2) ≤ 1))) then
           TWU_i = TWU_i^upper;      <The maximum TWU_i found.>
     else
     begin
           A_3 = LUF({h^j/p^j}, PDSA);
           A_4 = MF({h^j/p^j}, PDSA);
           <Phase 3>
           TWU_i = max(LP(A_1, PDSA), LP(A_2, PDSA), LP(A_3, PDSA), LP(A_4, PDSA));
           if (TWU_i = 0) then
              Call exception handler;
     end;
  end;
end;
```

Figure 2: The allocate algorithm.

algorithm is EDF, the available processor utilization is i (the number of non–faulty PE's).

When the scheduling algorithm is RM on the other hand, it is difficult to calculate the available processor utilization because it depends on the number of clones assigned to each PE. We assume that the number of clones assigned to a PE can be approximated by the average number of clones per PE, c/i where c is $\sum_{j=1}^{n} c_i^j$. The available processor utilization can be approximated by $c(2^{i/c}-1)$. Hence, when the scheduling algorithm is RM, TWU_i^{upper} and the upper bound utilization of a task may not be absolute upper bounds. The greedy algorithm has two steps.

(1) Each clone in PTS_i is assigned with the processing time of its hard task. If the scheduling algorithm is EDF and the total utilization of these clones exceeds i, then there is no feasible task allocation and we can stop the algorithm. Otherwise, the greedy algorithm sorts the task in PTS_i in non–increasing order of w^j/c_i^j.

(2) The algorithm tries to assign an additional s^j processing time to each clone of the first task T_i^j in the sorted task sequence. If the addition of s^j causes the total utilization of all clones to exceed the available processor utilization, then the algorithm tries to assign as much processing time as possible to each clone of T_i^j.[1] In this case, we stop the algorithm. Otherwise, the algorithm repeats the second step for the next task in the sequence.

When the scheduling algorithm is EDF, the greedy algorithm is optimal for solving the relaxed sub–problem due to the following facts. The processing time assigned to each clone in the first step is necessary for a feasible task allocation. The remaining problem after the first step is equivalent to the fractional knapsack problem [10], and the algorithm described in the second step is optimal. The complexity of the greedy algorithm is $O(n \log n)$.

Phase 2: LUF Algorithm and MF Algorithm

In the second phase, the allocate algorithm calls the *LUF* (largest utilization first) algorithm and the *MF* (multifit) algorithm to generate two task allocations for PTS_i using the upper bound u_i^j generated in the first phase. LUF has two steps.

(1) It sorts the tasks of PTS_i in non–increasing order of utilization. When multiple tasks have the same utilization, they are arranged in non–increasing order of the number of clones.

(2) LUF assigns the c_i^j clones of the first task T_i^j in the ordered task sequence to the c_i^j PE's that have the least excesses (or most slacks) in processor utilization. The excess of a PE is its current processor utilization minus its schedulability threshold and may be less than zero. Note that when the scheduling algorithm is RM, its threshold is anticipated as $(c+1)(2^{1/(c+1)}-1)$ where c is the number of clones currently assigned to it. LUF repeats the second step for the next task in the sequence.

The MF algorithm uses a first–fit bin–packing heuristic to minimize the maximum processor utilization. MF has three steps.

(1) It sorts the tasks of PTS_i in non–increasing order of the number of clones. If tie, the tasks are arranged in non–increasing order of utilization. MF sets the maximum processor utilization to the value, $\sum_{j=1}^{n} c_i^j u_i^j / m$.

---

[1] Specifically, the processing time allocated to each clone of T_i^j is determined by the remaining processor utilization times p^j divided by c_i^j. The remaining processor utilization is the available processor utilization minus the total utilization of all clones.

(2) MF assigns the c_i^j clones of the first task T_i^j in the ordered task sequence to the c_i^j PE's that fit with the lowest indices. It repeats the second step for the next task in the sequence.

(3) If MF fails to find a feasible packing with the maximum processor utilization, it increases the maximum processor utilization to one half of the current maximum processor utilization plus $4/3+(c_i-2)/3i$ where c_i is $\max_{j=1}^{n} c_i^j$. It goes to the second step. If a feasible packing with the maximum processor utilization is found, MF performs a binary search for the minimum maximum processor utilization in the interval between the current maximum processor utilization and the previous maximum processor utilization.

The interval to be searched in step (3) is between the lower bound value ($\sum_{j=1}^{n} c_i^j u_i^j/m$) and $4/3+(c_i-2)/3i$ where c_i is $\max_{j=1}^{n} c_i^j$. This is because we can show (Theorem 1 in Section 5.1) that the maximum processor utilization of LUF is bounded from above by $4/3+(c_i-2)/3i$, when the scheduling algorithm is EDF. When MF generates a maximum processor utilization larger than $4/3+(c_i-2)/3i$, the maximum processor utilization is larger than that of LUF. When the scheduling algorithm is RM, the upper bound is lower. Notes that when $c_i^j=1$ for all tasks, MF becomes the generalized FF version of the multifit algorithm.

When the scheduling algorithm is EDF and when either of the two maximum processor utilization associated with the task allocations A_1 and A_2 is 1 or less, TWU_i^{upper} is optimal. In the inequality case, $\sum_{j=1}^{n} c_i^j \tau^j/p^j < m$. Each soft task is executed to completion and there is idle processing time. We can stop the allocate algorithm. Since the upper bound task utilizations are used, the excesses of some PE's are usually positive. We define the *overage* of a task allocation as the largest excess of all PE's. We shall explain later in Section 6 that minimizing overage is consistent with maximizing *TWU*.

Because LUF and MF are heuristic in nature, they may generate infeasible task allocations. To increase the likelihood of getting feasible task allocations, the allocate algorithm also calls LUF and MF withe the hard subtasks of the clones only. Since LUF and MF are effective and h^j is usually less than τ^j, LUF and MF usually can find feasible task allocations for PTS_i. The complexity of LUF is $O(n\log n)$. The complexity of MF is $O(n\log n+in)$ where i is the number of iterations performed in the binary search and is usually preset to a constant.

Phase 3: Linear Programming Formulation

LUF and MF may generate inferior or infeasible task allocations, since the upper and lower bound utilizations of a task are used. Some processor utilizations in the task allocations may be greater than the schedulability thresholds (over–utilized) and others less than the thresholds

(under–utilized). We need to adjust u_i^j so that the utilization of every PE is close to but not more than the threshold and TWU_i is maximized. Specifically, the utilizations of some tasks allocated to the over–utilized PE's must be reduced, while some task utilizations should be increased in the under–utilized PE's. The problem of maximizing TWU_i can be formulated as an LP shown below in matrix notation.

$$\max \mathbf{w}\,\mathbf{u}$$
$$\mathbf{A}\,\mathbf{u} \leq \mathbf{threshold}(\mathbf{A}, PDSA)$$
$$\mathbf{h/p} \leq \mathbf{u} \leq \mathbf{r/p}$$

$\mathbf{w}$ is $1 \times n$ row vector of w^j and $\mathbf{u}$ $n \times 1$ column vector of u_i^j. Each task allocation can be described by a matrix $\mathbf{A}$ of $\{0, 1\}$ with i rows and n columns. An 1 at the j^{th} column and the k^{th} row denotes that a clone of the task T_i^j is assigned to the k^{th} PE. $\mathbf{threshold}(\mathbf{A}, PDSA)$ is a $i \times 1$ column vector of the schedulability thresholds for i PE's for the PDSA scheduling algorithm. $\mathbf{h/p}$ and $\mathbf{r/p}$ are the notations for $n \times 1$ column vectors of h^j/p^j and r^j/p^j respectively. We use known efficient algorithms for LP to compute the TWU_i for each task allocation [12, 23]. The allocate algorithm returns the largest TWU_i of the four task allocations. These algorithms solves our LP problem in $O(n^3)$ steps. The overall complexity of the allocate algorithm is thus $O(n^3)$.

An Example

We illustrate the allocate algorithm by an example. An instance of the replicated imprecise task allocation problem has 10 periodic tasks whose parameters are shown in Table 1. The real–time computer is a multiprocessor consisting of 6 PE's. We apply the allocate algorithm to find a task allocation for the sub–problem instance corresponding to operational mode 6. The scheduling algorithm employed is EDF. The second and third columns of Table 2 show the upper bound x_6^j and u_6^j produced by the greedy algorithm for each task T^j. The upper bound total weighted utilization TWU_6^{upper} is 13.493. When LUF is called with u_6^j, it generates the best of the four task allocations. Table 3 shows the task allocation matrix ($\mathbf{A}_1$) generated and the processor utilization of each PE. To compute TWU_6, we formulate and solve the problem as an LP. The efficient LP algorithm adjusts the processing time and utilization of each task as shown in Table 2. Every PE except PE 2 is fully utilized after the adjustment (Table 3). TWU_6 is 12.3533.

Table 1: An example of periodic task system

j	p	τ	h	w	c_6	c_5	c_4	c_3	c_2	c_1
1	20	4	2	5	3	2	2	1	1	0
2	30	5	3	9	3	3	3	2	2	1
3	50	15	2	4	3	2	1	1	1	0
4	50	20	11	8	5	4	3	2	2	1
5	100	29	10	10	4	4	3	3	2	1
6	100	41	20	2	2	1	1	0	0	0
7	200	47	36	7	2	2	2	2	1	1
8	200	90	25	3	1	1	1	1	0	0
9	300	78	49	1	1	1	1	0	0	0
10	300	79	33	6	4	3	2	2	1	1

Table 2: Upper bound utilizations given by the greedy algorithm and the utilizations given by LP and their corresponding assigned processing times

j	Greedy Algorithm		LP	
	x_6	u_6	x_6	u_6
1	4	0.200000	2.56666	0.128333
2	5	0.166667	5	0.166667
3	2	0.040000	2	0.040000
4	16.966667	0.339333	18	0.360000
5	29	0.290000	29	0.290000
6	20	0.200000	20	0.200000
7	47	0.235000	47	0.235000
8	90	0.450000	89.6666	0.448333
9	49	0.163333	66.4999	0.221667
10	33	0.110000	33	0.110000

Table 3: The task allocation matrix (A_1) generated by LUF and the processor utilizations

	j										*Processor Utilization*	
PE	1	2	3	4	5	6	7	8	9	10	Greedy Alg.	LP
1	0	1	1	0	0	0	1	1	0	1	1.001667	1
2	1	1	0	1	1	0	0	0	0	0	0.996000	0.945
3	1	0	0	1	1	0	0	0	1	0	0.992667	1
4	0	0	1	1	1	1	0	0	0	1	0.979333	1
5	0	0	1	1	1	1	0	0	0	1	0.979333	1
6	1	1	0	1	0	0	1	0	0	1	1.051000	1

5. Performance Evaluation

The allocate algorithm is evaluated using stochastic analysis techniques and Monte Carlo simulations. We focus the analytical evaluation on the LUF algorithm where the scheduling algorithm is EDF. A sub–problem of a replicated imprecise task allocation problem can have many task allocations. Associated with each task allocation is a TWU_i. Suppose there are two task allocations for a sub–problem. In general but not always, the task allocation with a smaller overage tends to have a higher TWU_i than the task allocation with a larger overage. A larger overage means that the processor utilization of some PE in a task allocation exceeds its schedulability threshold by a greater amount. When the scheduling algorithm is EDF, the overage of a task allocation is its maximum processor utilization minus one. It is more difficult for an LP algorithm to reduce the larger overage to zero so that the task allocation is feasible and without compromising TWU_i. Hence, the task allocation with a smaller overage is preferred. We show that the ratio $R(LUF)$ between the maximum processor utilization produced by LUF and the optimal (minimum) maximum processor utilization for mode i is bounded from above by $4/3+(c_i-2)/3i$ in the worst case, where $c_i=\max_{j=1}^{n} c_i^j$. As the number of tasks increases, the maximum processor utilization generated by LUF approaches the optimal maximum processor utilization asymptotically. The latter result is derived under a mild assumption on the probability distribution (F) of the utilizations of the periodic tasks. Due to the space limitation, we state the following theorems without proofs [26].

5.1. Bounds on the Maximum Processor Utilization of LUF

LUF algorithm sorts the tasks in PTS_i so that for all $k>j$, either $u_i^k>u_i^j$, or $u_i^k=u_i^j$ and $c_i^k \geq c_i^j$. Let $Z_k^j(LUF)$ denote the processor utilization of the k^{th} PE after the LUF algorithm have scheduled the clones of task T^j. We use $Z_{(1)}^j(LUF)$ and $Z_{(i)}^j(LUF)$ to denote the PE's with the smallest and largest processor utilizations respectively in the ordered sequence of $\{Z_k^j(LUF) \mid k=1, 2, \ldots, i\}$. $R(LUF)$ is defined as $Z_{(i)}^n(LUF)/Z_{(i)}^n(OPT)$ where $Z_{(i)}^n(OPT)$ is the optimal maximum processor utilization.

Theorem 1: $R(LUF) \leq 4/3+(c_i-2)/3i$ where $c_i=\max_{j=1}^{n} c^j$.

In the example shown in Table 1, $c_6=5$ and $R(LUF) \leq 3/2$ for mode 6. $R(LUF)$ is always less than or equal to 5/3 because $c_i \leq i$. Let $D^j(LUF)$ denote the difference between the largest processor utilization and the average processor utilization after the clones of the task T^j have been allocated by the LUF algorithm. $D^j(LUF)=Z_{(i)}^j(LUF)-\sum_{k=1}^{i} Z_k^j(LUF)/i$.

Theorem 2: If the expected utilization Eu is finite and the probability distribution F of u_i^j is strictly increasing on $(0, \epsilon)$ for some $\epsilon > 0$, then

$$\lim_{n \to \infty} D^n(LUF) = 0. \text{ (a.s.)}$$

Corollary 3: If the expected utilization Eu is finite and the probability distribution F of u_i^j is strictly increasing on $(0, \epsilon)$ for some $\epsilon > 0$, then

$$\lim_{n \to \infty} (Z_{(i)}^n(LUF) - Z_{(i)}^n(OPT)) = 0. \text{ (a.s.)}$$

Proof: The proof is similar to that in the reference [8]. □

5.2. Simulation Results

These analytic results do not show how large an n will make TWU_i close to the optimal, and how TWU_i is affected by RM, the variations in the hard tasks, and the number of PE failures. Furthermore, the analytic results do not show how these variations cause the allocate algorithm to produce infeasible task allocations. We address these questions by simulations next.

The task systems are generated randomly for each simulation. For the lack of prior knowledge, random variables are assumed to be uniformly distributed. The number c_m^j of clones for each task T_m^j in PTS_m is uniformly distributed in the interval between 1 and m inclusively $[1, m]$. So, T_m^j has an average of $\bar{c}_m (= (1+m)/2)$ clones. The average workload of a hard task $\bar{u}_h = \bar{h}/\bar{p}$ is half of the average task workload $\bar{u} = \bar{\tau}/\bar{p}$. Hence, the average hard workload of PTS_m is $n\bar{c}_m \bar{u}_h$. We define the average ***normalized hard workload*** (of PTS_m) as $n\bar{c}_m \bar{u}_h / m$.

We use the following three steps to generate iteratively the number of clones of each task for each operation mode. Suppose for each task T_i^j the number c_i^j of clones has been determined and we want to generate c_{i-1}^j. First, we choose a random number uniformly distributed in $[1, n]$ and want to delete this number of clones from PTS_i. Second, for each task T_i^j if $c_i^j > i-1$, then $c_{i-1}^j = i-1$. T_{i-1}^j has one clone less than T_i^j. If the total number of clones deleted in this way is more than or equal to the random number chosen in the first step, then we stop and c_{i-1}^j has been determined. Otherwise, let d be the difference. Third, we select d tasks uniformly from PTS_i and delete one clone from each. A task can be selected multiple times. If a selected task has zero clone, then no clone is deleted. Initially, PTS_m has an average of $n(m+1)/2m$ clones per PE. We remove $(n+1)/2$ clones on the average from PTS_i to get PTS_{i-1}. As the number of non–faulty PE's is reduced to 1, the average number of clones per PE is increased to

$n-(m-1)/2$. Thus, we simulate the unwillingness to delete the clones from a task system and the increase in the hard workload, as more PE's fail.

The sample size at each data point is 20. The number (m) of non–faulty PE's is set to 20 initially for each simulation. To completely evaluate the allocation algorithm for a replicated imprecise task allocation problem, we need to execute the allocate algorithm m times to solve m sub–problems. Hence, 400 sub–problems are evaluated at each data point.

The performance of the allocate algorithm is measured by two metrics. The first metric is the average ratio $\theta = TWU_i/TWU_i^{upper}$. θ is averaged over the feasible sub–problems at each data point. The upper bound value is used instead of the optimal value because the problem is NP–hard. The ratio θ is a conservative measurement in the sense that TWU_i^{upper} is always (respectively, usually) greater than the optimal value, when the scheduling algorithm is EDF (respectively, RM). The second metric is the number of infeasible task allocations generated at each data point. As discussed earlier, the allocate algorithm is heuristic in nature and may generate infeasible task allocations for feasible sub–problems. This metric shows how good the algorithm is in finding feasible solutions. This metric is also conservative. For some task systems with high hard workloads, there are no feasible task allocations. However, the greedy algorithm solves the relaxed problem in which two constraints for feasible task allocations are ignored. It may falsely declare some sub–problems feasible that are in fact infeasible.

5.2.1. Effect of the Number of Tasks

We are interested in knowing how large n must be to make θ close to 100%. As we increase the number of tasks in a task system, the hard workload of the task system increases. To ensure that the task systems generated are not biased against large numbers of tasks and that few infeasible task systems are randomly generated, we keep the normalized hard workloads ($n\bar{c}_m\,\bar{u}_h/m$) of task systems constant and less than 1. This can be done by setting the average hard workload ($\bar{u}_h$) of a task equal to $1/n$. Hence, the utilization range (u_m^j) of a task T_m^j is $(0, 4/n)$. In this simulation, the average normalized hard workload is fixed at 0.5. (More precisely, $\bar{c}_m/m$ =0.5125.) Figure 3 shows that θ approaches to 100% monotonically, as the number of tasks increases and the average hard workload ($\bar{u}_h$) decreases. When the number of tasks is 40, $\theta(EDF)$ and $\theta(RM)$ are 99.8% and 99.63% respectively. The result is implied indirectly by Theorem 2. The second column of Table 4 lists the numbers of infeasible sub–problems found by the greedy algorithm, when PDSA is EDF. When the number of tasks is 10, there are 39 infeasible sub–problems. The third and fourth columns of Table 4 displays the numbers of infeasible

task allocations generated, when PDSA is EDF and RM respectively. In EDF case, all task allocations are feasible except one.

5.2.2. Effect of Hard Tasks

It is more difficult to schedule a task system with a higher normalized hard workload. In this simulation, we study its effect on θ and the number of infeasible task allocations. We let the

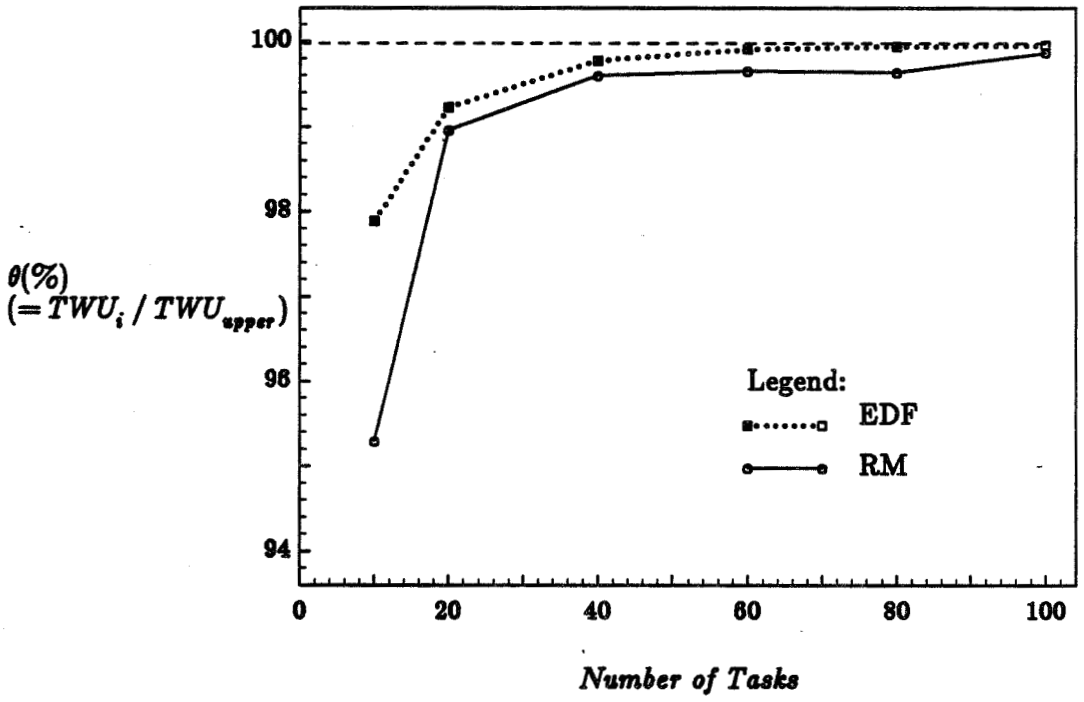


Figure 3: Effect of the number of tasks on *TWU*.

Table 4: Effect of the Number of Tasks

Number of Tasks	*Infeasible Sub–problems Greedy Alg.(EDF)*	*Infeasible Task Allocations*	
		EDF	*RM*
10	39	1	68
20	0	0	71
40	0	0	56
60	0	0	73
80	0	0	78
100	0	0	43

number of tasks be 60 and increase the normalized hard workload from 0.375 to 1. When PDSA is EDF, Table 5 shows that the number of infeasible sub–problems found by the greedy algorithm increases monotonically. In contrast, the allocate algorithm produces four infeasible task allocations. When PDSA is RM, the number of infeasible task allocations increases monotonically. If the normalized hard workload is greater than 0.75, most task allocations are infeasible. As the workload of the hard tasks in a task system gets higher, Figure 4 shows that θ(EDF) and θ(RM) decrease monotonically. The worst case $\theta(EDF)$ is still greater than 98% and $\theta(RM)$ 95%.

Table 5: Effect of Hard Tasks

Normalized Hard Workload	*Infeasible Sub–problems Greedy Alg.(EDF)*	*Infeasible Task Allocations*	
		EDF	*RM*
0.375	0	0	3
0.5	0	0	73
0.625	44	0	210
0.75	81	1	319
0.875	169	3	389
1.0	307	0	

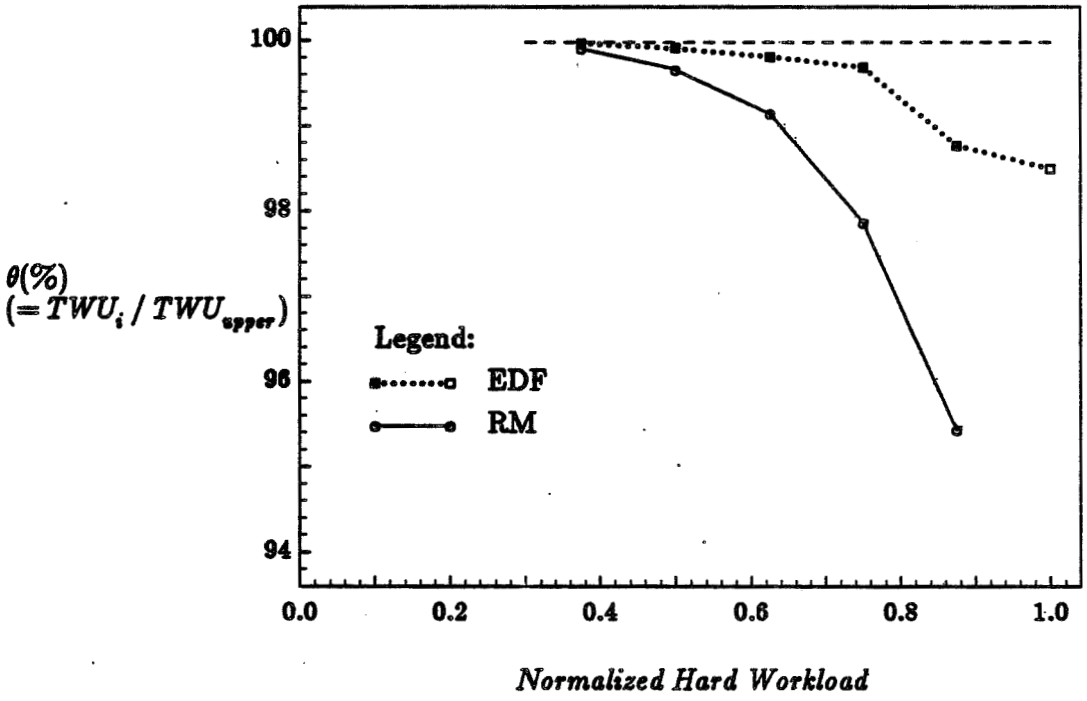


Figure 4: *TWU* and the hard workload.

5.2.3. Effect of Hardware Failures

Finally, we show the effect of hardware component failures. We use the samples in Section 5.2.1, and average and present them according to the number of PE failures in stead of the number of tasks. Figure 5 shows that θ has a decreasing trend, as the number of PE failures increases. The zigzagging curve may be explained by the probabilistic manner used to delete the clones from PTS_i. The normalized hard workload of PTS_i may be greater than that of PTS_{i-1}. The rising tail curve is because TWU_i of infeasible task allocations are excluded from θ. When the number of fault–free PE is one, task allocations are either infeasible or optimal. Figure 6 shows that the number of feasible task allocations (RM) tends to decrease, as more PE's become faulty and the normalized hard workload approaches one, exceeding the RM schedulability threshold. Whereas the normalized hard workload is less than the EDF schedulability threshold, the number of feasible task allocations (EDF) stay constant.

6. Conclusion

We have presented the allocate algorithm for scheduling real–time replicated tasks under hardware failures. When hardware components fail, less computational resources are available. The imprecise computation model provides the scheduling algorithm with the flexibility of trading off the quality of the results with the amount of computational resources required to produce them. Furthermore, when less computational resources are available, real–time systems may require a fewer number of clones to perform critical functions. As a result, these two flexibilities permit the performance of the real–time system to degrade gracefully under hardware failures.

We have evaluated the allocate algorithm by stochastic analysis and simulations. The simulation results show that the algorithm are effective under hardware failures. The allocate algorithm oftentimes generate (almost) optimal solutions, while it seldom produces infeasible task allocations. The analytical results show that when the priority driven scheduling algorithm used is EDF, the LUF algorithm produced task allocations whose maximum processor utilizations approach the optimal asymptotically, as the number of tasks increases. The analytical results also show that the ratio of the worst–case maximum processor utilization and the optimal maximum processor utilization is bounded by a small number.

References

[1] T. Anderson and J. C. Knight, "A Framework for Software Fault Tolerance in Real–Time Systems," *Trans. on Software Eng.*, vol. SE–9, no. 3, pp. 355–364, IEEE, May 1983.

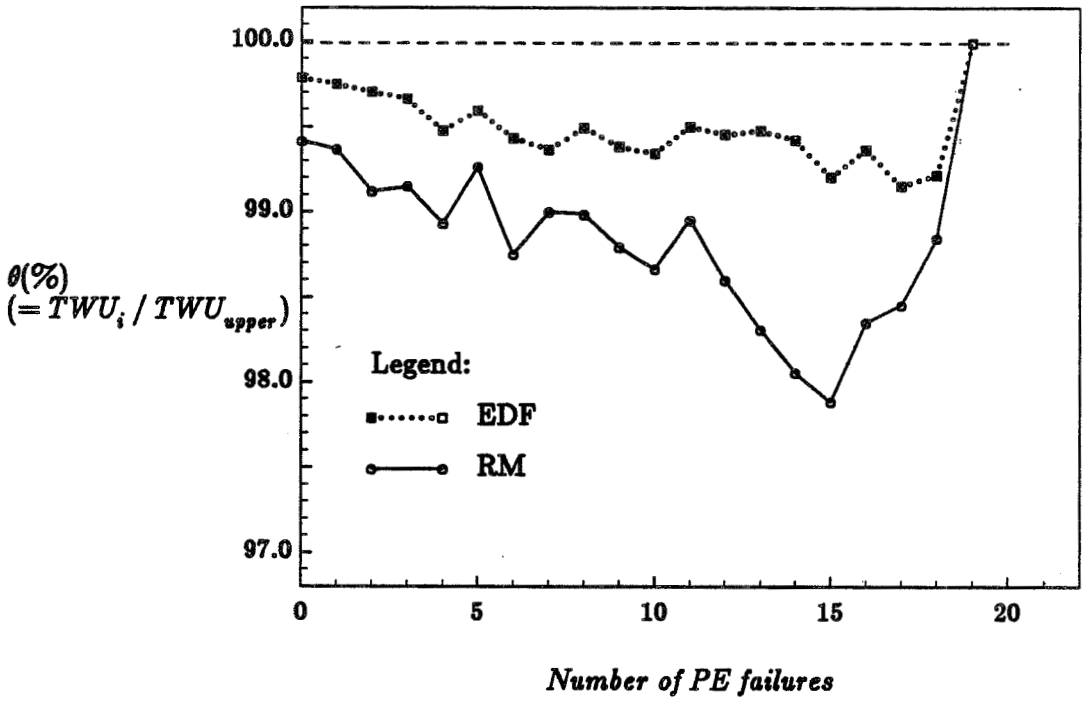


Figure 5: *TWU* and PE failures.

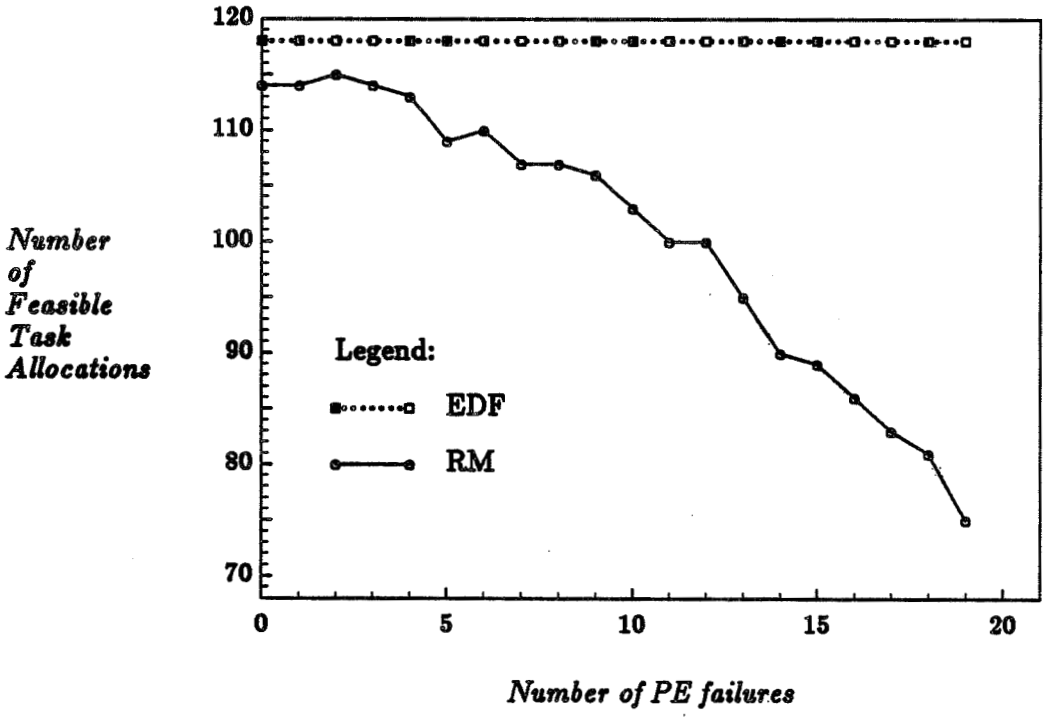


Figure 6: Feasible Task Allocations and PE failures.

[2] A. Avizienis and J. C. Laprie, "Dependable Computing: From Concept to Design Diversity," *Proc. of the IEEE*, vol. 74, no. 5, pp. 629–638, May 1986.

[3] J. Bannister and K. Trivedi, "Task Allocation in Fault–Tolerant Distributed Systems," *Acta Informatica*, vol. 20, pp. 261–281, 1983.

[4] F. Christian, "Understanding Fault–Tolerant Distributed Systems," *CACM*, vol. 34, no. 2, pp. 56–78, 1991.

[5] J. Y. Chung, J. W. Liu, and K.–J. Lin, "Scheduling Periodic Jobs That Allow Imprecise Results," *Trans. on Computers*, vol. C–39, no. 9, pp. 1156–1174, IEEE, Sep. 1990.

[6] S. K. Dhall and C. L. Liu, "On a Real–Time Scheduling Problem," *Oper. Res.*, vol. 26, no. 1, pp. 127–140, Jan. 1978.

[7] P. Ezhilchelvan and R. de Lemos, "A Robust Group Membership Algorithm for Distributed Real–Time Systems," *Proc. 11th Real–Time Systems Symp.*, pp. 173–179, IEEE, Dec. 1990.

[8] J. B. Frenk and A. H. G. Rinnooy Kan, "The Asymptotic Optimality of the LPT Rule," *Math. Oper. Res.*, vol. 12, no. 2, pp. 241–254, May 1987.

[9] Michael R. Garey and David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP–Completeness*, Freeman, San Francisco, CA, 1979.

[10] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Rockville, MD, 1978.

[11] R. Jurgen, "Smart Cars and Highways Go Global," *Spectrum*, pp. 26–36, IEEE, May 1991.

[12] N. Karmarkar, "A New Polynomial–time Algorithm for Linear Programming," *Combinatorica*, vol. 4, pp. 373–395, 1984.

[13] K. H. Kim and H. O. Welch, "Distributed Execution of Recovery Block: An Approach for Uniform Treatment of Hardware and Software Faults in Real–Time Applications," *Trans. on Computers*, vol. C–38, no. 5, pp. 626–636, IEEE, May. 1989.

[14] C. M. Krishna and K. G. Shin, "On Scheduling Tasks with a Quick Recovery from Failure," *Trans. on Computers*, vol. C–35, no. 5, pp. 448–454, IEEE, May 1986.

[15] J. Lehoczky, "Fixed Priority Scheduling of Periodic Task Set with Arbitrary Deadlines," *Proc. 11th Real–Time Systems Symp.*, pp. 201–209, IEEE, Dec. 1990.

[16] K.–J. Lin, S. Natarajan, and J. W. Liu, "Imprecise Results: Utilizing Partial Computations in Real–Time Systems," *Proc. 8th Real–Time Systems Symp.*, pp. 210–217, IEEE, Dec. 1987.

[17] C. L. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real–Time Environment," *JACM*, vol. 10, no. 1, pp. 46–61, Jan. 1973.

[18] R. Rajkumar, "Real–Time Synchronization Protocols for Shared Memory Multiprocessors," *Proc. 10th Int'l Conf. on Distributed Computing Systems*, pp. 116–123, IEEE, May 1990.

[19] K. Ramamritham, "Allocation and Scheduling of Complex Periodic Tasks," *Proc. 10th Int'l Conf. on Distributed Computing Systems*, pp. 108–115, IEEE, May 1990.

[20] W. K. Shih, J. W. Liu, and J. Y. Chung, "Fast Algorithms for Scheduling Imprecise Computations," *Proc. 10th Real–Time Systems Symp.*, pp. 12–19, IEEE, Dec. 1989.

[21] K. G. Shin, C. M. Krishna, and Y. H. Lee, "A Unified Method for Evaluating Real–Time Computer Controllers and Its Applications," *Trans. on Automat. Contr.*, vol. AC–38, no. 4, pp. 357–366, IEEE, Apr. 1985.

[22] B. Sprunt, J. Lehoczky, and L. Sha, "Exploiting Unused Periodic Time for Aperiodic Service Using the Extended Priority Exchange Algorithm," *Proc. 9th Real–Time Systems Symp.*, pp. 251–258, IEEE, Dec. 1988.

[23] P. Vaidya, "An Algorithm for Linear Programming which Requires $O(((m+n)n^2+(m+n)^{1.5}n)L)$ Arithmetic Operations," *Math. Prog.*, no. 47, pp. 175–201, 1990.

[24] J. Wensley, L. Lamport, J. Goldberg, M. Green, K. Levitt, P. Melliar–Smith, R. Shostak, and C. Weinstock, "SIFT: Design and Analysis of a Fault–Tolerant Computer for Aircraft Control," *Proc. of the IEEE*, vol. 66, no. 10, pp. 1240–1255, Oct. 1978.

[25] A. C. Yu and K.–J. Lin, "Scheduling Parallelizable Imprecise Computations on Multiprocessors," *Proc. 5th Int'l Parallel Processing Symp.*, pp. 531–536, IEEE, Apr. 1991.

[26] A. C. Yu and K.–J. Lin, "A Multiprocessor Scheduling Algorithm for Fault–Tolerant Imprecise Computations," Department of Computer Science Technical Report UIUCDCS–R–91–1691, University of Illinois at Urbana–Champaign, Urbana, IL, May. 1991.

NASA S2-61
93848
245133
10 P.

An Object–Oriented Query Processor that Produces Monotonically Improving Approximate Answers *

S. V. Vrbsky and J. W. S. Liu

Department of Computer Science
University of Illinois, Urbana, IL 61801
{vrbsky, janeliu}@m.cs.uiuc.edu

Abstract

This paper describes an object–oriented query processor that makes approximate answers available if there is not enough time to produce an exact answer or if part of the database is unavailable. The accuracy of the approximate result produced improves monotonically with the amount of data retrieved to produce the result. The query processing algorithm is based on an approximate relational data model and works within a standard relational algebra framework. The query processor maintains an object–oriented view on an underlying level and can be implemented on a relational database system with little change to the relational architecture. We show how a monotone query processing strategy can be implemented, making effective use of semantic information presented by the object–oriented view.

1. Introduction

For many applications, it may be necessary for a database to produce an answer even when part of the database is unavailable. In a hard real–time environment, such as avionics, air traffic control, and multiple robot systems, a result produced too late is of little or no use. When there is not enough time to retrieve all the data needed to answer a query, producing an approximate answer in time is often better than having no answer. Making a usable, *approximate result* available, when the exact one cannot be produced in time, is the basic strategy taken in the imprecise computation approach [1–2]. This approach provides scheduling flexibility and supports graceful degradation in hard real–time environments. Other factors, such as a network partition or component failures, may cause some needed data to become inaccessible. Making approximate answers to database queries available under such circumstances is one way to enhance the fault tolerance and availability of a database system.

Smith and Liu [3] have developed an approximate query processing algorithm that derives approximate answers to set–valued queries from the query expression. This algorithm is monotone in the following sense: it produces approximate answers which are non–decreasing in accuracy with an increasing time spent, and hence the amount of data retrieved, to produce them. The algorithm works within the framework provided by a standard relational algebra query language and is based on an approximate relational data model. The model defines for any standard relation a set of approximate relations, a partial–order relation over this set for comparing the approximate relations, and a complete set of approximate relational algebra operations with approximate relations as operands. An initial approximation of the exact answer is computed when query processing begins. This approximate answer is improved according to the partial–order relation as more data is retrieved to answer the query. The exact answer is returned if query processing is allowed to continue until completion. The latest, best available approximate answer is returned if query processing must be terminated before it is completed.

This paper describes an object–oriented query processor to efficiently implement such a monotone query processor. The semantic information provided by the object–oriented model, within the view maintained by the query processor, makes it possible to reduce the computation time needed to generate approximate relations and to perform approximate relational algebra operations. The object–oriented query processor derives approximate answers to

* This work was partially supported by NASA Contract No. NASA NAG 1–613 and AFOSR Contract No. 90 0193.

In *Proceedings of IEEE International Conference on Data Engineering, Kobe, Japan, April 1991.*

standard relational algebra queries. The data is stored as relations. As shown in Figure 1, this query processor can be implemented on a relational database system, requiring little or no change to the underlying relational architecture.

The work on monotone query processing described here assumes that the database is complete and the query is precise. The imprecision in the query response occurs either because of the time factor, there is not enough time to complete query processing by a deadline, or the data is unavailable. In contrast, much of the current research on imprecision in databases is concerned with databases that contain incomplete data [4–6]. In an incomplete database, the value for an attribute may be null, partial or uncertain (fussy) [7]. There are methods for the representation and access of such data in a relational database model. There has also been research on vague queries [8], in which a query is expressed in terms of qualifications that are imprecise. The response to a vague query is data which closely matches the qualifications of the query. Query processing that produces monotonically improving answers has been proposed by Buneman, Davidson, and Watters [9,10]. Their method uses a set of static rules to represent the relationships among the data in order to produce monotonically improving approximate answers.

The remainder of this paper is organised as follows. Section 2 describes the approximate relational model. Section 3 describes the design of the object–oriented query processor. Section 4 describes how relational algebra queries are processed by the object–oriented query processor. Section 5 suggests further extensions and presents conclusions.

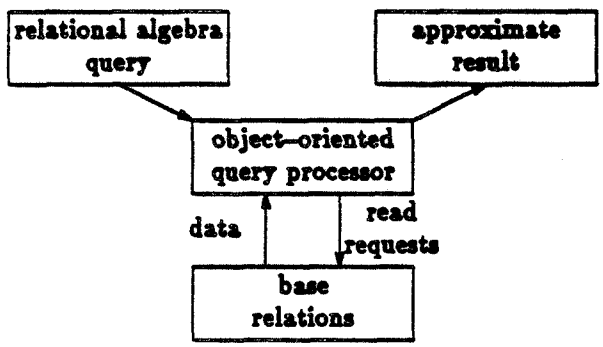


Figure 1. Object–oriented query processor

2. Approximate Relational Data Model

Given a standard relation S in the traditional relational model, all the tuples in a subset of S certainly belong to S, and a tuple in a superset of S is possibly also a tuple in S. Therefore, an approximation of any standard relation S can be defined in terms of a subset and a superset of S [3]. This approximation is called an *approximate relation*. Formally, an approximate relation R of a relation S is composed of two blocks of tuples, a set C of tuples whose memberships in S are certain, and a set P of tuples whose memberships in S are uncertain, such that $C \subseteq S$ and $R = C \cup P \supseteq S$. We refer to tuples in these two blocks as *certain* tuples and *possible* tuples, respectively. An approximate relation R is denoted by the 2–tuple (C, P).

As an illustrative example, suppose that a salesman queries an inventory database in order to locate all the Ford, 4–door, red sedans his used car dealership has on hand. The exact answer to his query is the standard relation S shown in Figure 2(a). Figures 2(b) – (f) show five other relations. The relation R_1 shown in Figure 2(b) contains tuples on all Ford, 4–door, sedans. It is a superset of S, containing tuples that are possibly in S and, hence, can be considered an approximation of S. The relation R_2 shown in Figure 2(c) contains tuples on all Ford, 4–door, red sedans located at Champaign. It is a subset of S, containing tuples that are certainly in S. For a reason that will become evident later, we do not consider R_2 an approximation of the standard relation S. R_3 shown in Figure 2(d) is another approximation of S. The first three tuples are certainly in S. They form a subset of S. The last three tuples are possible ones in S. R_3 is a superset of S. It is a subset of R_1, and, as will be explained later, it is a better approximation of S than R_1. The relation R_4 in Figure 2(e), another superset of S, is the set of tuples on all Ford, red sedans. It is a different superset of S than R_1. The approximate relation R_5 in Figure 2(f) is a subset of R_4 and a superset of S.

As illustrated by the example above, any standard relation S has many approximate relations. Given a set of approximate relations of S, a partial order relation $\geq$ for comparing them can be defined over the set. One approximate relation $R_i = (C_i, P_i)$ is better than or equal to another $R_j = (C_j, P_j)$, denoted as $R_i \geq R_j$, if $C_i \supseteq C_j$, $P_i \subseteq P_j$, and $C_i - C_j \subseteq P_j - P_i$. The tuples in $C_i - C_j$ are considered to have migrated from the possible part of

R_j into the certain part of R_i as the approximation improves, and the remaining tuples in $P_j - P_i$ are deleted. The partial ordered set of all approximate relations of S is a lattice. In the lattice, $R_0 = (\emptyset, \upsilon)$ is the least element and the worst possible approximation of S, where $\emptyset$ denotes the null set and υ is the set of all possible tuples which could be in S, the cartesian product of all the domains in the schema of S. We assume that υ is finite. The greatest element of the lattice is the best possible approximation of S and is S itself, which is represented by $(S, \emptyset)$.

We consider again the approximate relations in Figure 2. $R_3 \geq R_1$ because in R_3 there are more certain tuples and less possible ones than in R_1. $R_5 \geq R_4$ for the same reason. Although both R_3 and R_5 are better approximations of S than R_1 and R_4, respectively, we cannot say whether R_3 is better than R_5 or vice versa. This is because R_3 and R_5 are not mutually comparable by the partial order defined above.

Car_No	Location
5421	Urbana
7824	Springfield
7924	Springfield
9021	Champaign
9122	Champaign

Figure 2(a). S: All Ford, 4-door, red sedans

Car_No	Location
5421	Urbana
5521	Urbana
5712	Urbana
7824	Springfield
7826	Springfield
7924	Springfield
9021	Champaign
9075	Champaign
9122	Champaign
9555	Champaign

Figure 2(b). R_1: All Ford, 4-door sedans

Car_No	Location
9021	Champaign
9122	Champaign

Figure 2(c). R_2: Champaign, Ford, 4-door, red sedans

Car_No	Location
5421	Urbana
9021	Champaign
9122	Champaign
7824	Springfield
7826	Springfield
7924	Springfield

Figure 2(d). R_3: Approximation.

Car_No	Location
5411	Urbana
5421	Urbana
7824	Springfield
7924	Springfield
7926	Springfield
9011	Champaign
9021	Champaign
9122	Champaign

Figure 2(e). R_4: All Ford, red sedans

Car_No	Location
5421	Urbana
7824	Springfield
7924	Springfield
9011	Champaign
9021	Champaign
9122	Champaign

Figure 2(f). R_5: Approximation.

It is not possible to apply standard relational algebra operations to approximate relations. Smith and Liu defined new approximate relational algebra operations: union, set difference, select, project, and cartesian product. The definitions of a complete set of approximate relational algebra operations for approximate relations are listed in Table 1 of Appendix A. Join can be derived from select and cartesian product, and intersection from set difference. In the definitions, $\times$, $\cup$, $\cap$, and $-$ in the C_T and P_T columns denote the set-theoretical operations cartesian product, union, intersection and set difference, respectively. Every approximate operation is monotone; the result of an operation is better when its operands are better.

The monotone query processing algorithm begins by first parsing a relational algebra query and representing it by a query tree. Each leaf node of the tree represents a read request to retrieve a base relation, and each non-leaf node represents a relational algebra operation. For example, the query:

$$R_T = \sigma_{\text{red-sedan}} R_s \cup (R_g \times R_s)$$

where σ is the select operator, is represented by the tree shown in Figure 3. When a node is evaluated, the result of the operation represented by the node is the value of the node. When the operands of a node are available, the node can be evaluated. The evaluation process begins at the leaf nodes, and repeats upward in the tree until the root node is evaluated. The value produced at the root node is the query answer. For monotone query processing, the relational algebra operation at every node of the query tree is an approximate one. Each node is assigned an initial approximate value. A read request at a leaf node can return segments of a base relation at a time rather than the entire relation. Tuples in the returned segments are certain tuples in an approximate relation of the base relation. As segments of a base relation are returned, the value of the leaf node improves. The improvement is propagated upward to the root node by reevaluating the nodes in the query tree. The value of the root node is repeatedly updated with better values, and a series of approximate answers are produced that increase in quality monotonically as more data is retrieved. If processing of the query is terminated

prematurely (due to a deadline or a network partition, for instance), the current value of the root is returned as the approximate answer to the query. Because the set of all approximate relations of every standard relation is a lattice, no matter what approximate relation is used as an initial approximation of the response to a query, the exact answer will be obtained if query processing is allowed to proceed until completion.

3. An Object–Oriented Query Processor

To implement monotone query processing efficiently, two problems must be solved. The first one is how to generate the approximate relations that are used as the initial approximations of the nodes in the query tree. A factor that affects the efficiency of monotone query processing is the number of possible tuples in each initial approximation. As stated above, theoretically, we can use the worst approximation of any relation as an initial approximation, whose possible set is the cartesian product of all the domains in the schema of the relation and whose certain set is empty. Such an initial approximation can be computed, and therefore, can be generated without reading any data. (Again, we assume here that all domains are finite.) Unfortunately, this approximate relation invariably contains a large number of possible tuples. The time required to delete and migrate possible tuples to produce new improved approximate relations can be made smaller by choosing better initial approximations. As an example, we consider again the query to locate all the Ford, 4–door, red sedans in Figure 2. Suppose that a relation containing tuples on all Ford cars is to be used as an initial approximation of the exact answer. This relation contains tuples on sports coupes and sedans, 2–door as well as 4–door sedans, cars of all colors, etc. In other words, it contains a large number of possible tuples which will have to be deleted and migrated to produce the exact answer. Better initial approximations are relations R_1 and R_4 in Figure 2. They can be used if we know how to identify them and generate them.

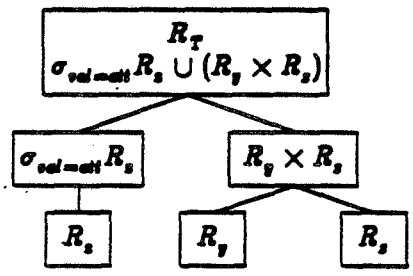


Figure 3. The tree representation of a query

The second problem we must solve is how to implement the approximate relational algebra operations themselves efficiently. The approximate relational algebra operations listed in Table 1 involve performing operations on the possible tuples, as well as the certain tuples, in their operands. Implementing monotone query processing in the manner described in Section 2 involves performing approximate relational algebra operations on possible tuples each time an improvement in the approximation of some base relation is propagated up the query tree. Query processing time can be saved if operations on the possible tuples could be avoided.

The approximate relational model itself does not provide us with sufficient semantic support to implement the model efficiently. We now demonstrate that an object–oriented approach to implementing the query processor provides us with practical solutions to these problems. We describe here an object–oriented, monotone query processor that relies on an object–oriented view to provide the semantic information needed for an efficient implementation of the monotone query processing algorithm. The object–oriented model, on an underlying level, offers solutions to the above mentioned problems while allowing the approximate relational data model to be maintained on the external level. In other words, the user's view remains that of the relational model. Moreover, the data is stored as relations. This allows the object–oriented monotone query processor to be implemented on a relational database system, requiring little or no change to the underlying relational architecture. The object–oriented query processor can be one of several query processors available in the database system.

3.1. Classes and Class Hierarchies

Similar to other object–oriented systems, our query processor is composed of objects and classes [11–13]. Classes are organised into class hierarchies; the child of a class is called its *subclass*, and the parent of a class is called its *superclass*. Currently, we have made use of only one kind of class hierarchy; in each class hierarchy, every subclass has a unique

superclass (that is, single inheritance) and inherits all the properties of its superclass. In other words, the relationship between a subclass and its superclass is one of IS–A. A subclass is considered a *specialization* of its superclass, and the superclass is considered the *generalization* of its subclasses. In particular, each superclass is partitioned by its subclasses; the sets of all instances of different subclasses are disjoint, and the set of all instances of a superclass is the set of all instances of all its subclasses. (Our plan to experiment with the use of other kinds of class hierarchies is described in Section 5.)

The query processor maintains, for its purposes, an object–oriented view of the relational database. It views each base relation, and sometimes each segment of the relation, as a class; tuples in the relation, or the segment, are instances of the corresponding class. Hereafter, we also refer to tuples stored in the database as *database objects*. A collection of class hierarchies is used to present a semantic interpretation of the database objects. A class hierarchy provides access path information. This information is accessed along with the base relations during query processing.

Specifically, each class in a class hierarchy represents a unit of stored data that can be read at one time during query processing. The information on this number is provided by the class hierarchy and is maintained by the query processor. The number of database objects returned by a read request varies. It is possible that some base relations can be read only in their entirety. A read request issued against such a base relation returns all of the database objects in the relation when it completes. This fact is indicated in a class hierarchy with a base relation being a class that has no subclasses. If a partial read request can be issued to retrieve individual segments of a base relation, the class corresponding to the base relation has subclasses, each corresponding to a segment. For example, Figure 4 shows a collection of class hierarchies representing an object–oriented view of the inventory database of a used car dealer. The relational database consists of two relations, CARS and DEALERSHIPS. The class hierarchy indicates that the CARS relation is stored so that the segment containing database objects on GM cars or 2–door Ford sedans can be retrieved by a partial read request.

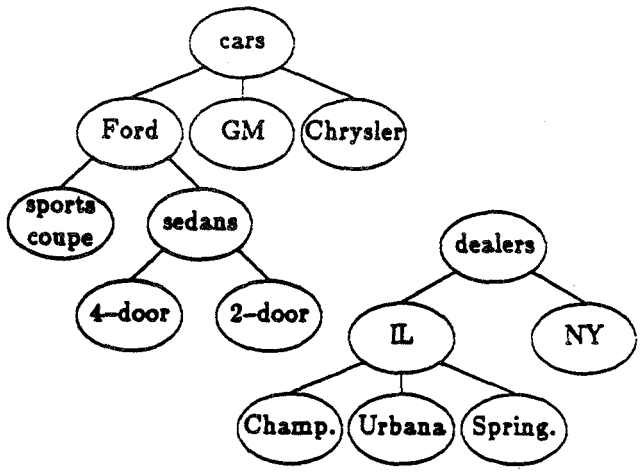


Figure 4: Class hierarchies

3.2. Approximate Class and Objects

The query processor maintains a set of classes, referred to as *approximate classes*, which are distinct from the classes in the class hierarchies described above. Each approximate class provides a description of a set of approximate relations of a standard relation S. An approximate class has three attributes, that is, instance variables: Certain_Part, Possible_Part and OP. The schema of an approximate class is illustrated in Figure 5(a). The domain of the attribute Certain_Part is the class C. Instances of C are the certain tuples in an approximate relation of S. We refer to the class C as the *certain class*. The domain of the attribute Possible_Part is a metaclass whose instances are themselves classes. We denote this set of instances that are classes by P. As illustrated by Figure 5(b), the instances of a class in the set P are the possible tuples in an approximate relation of S. We refer to the classes in P as *possible classes*. The attribute OP in the approximate class has as its value a relational algebra operator or read_request. We will return to discuss this attribute.

An instance of an approximate class is an *approximate object*. Each approximate object corresponds to an approximate relation. An approximate object R_i, corresponding to an approximate relation $R_i = (C_i, P_i)$, is described by the 2–tuple (C_i, P_i); tuples in the certain set C_i are instances of the certain class C, and P_i is a set of possible classes, the set of all instances of which is the possible set P_i. Let us consider again the example in

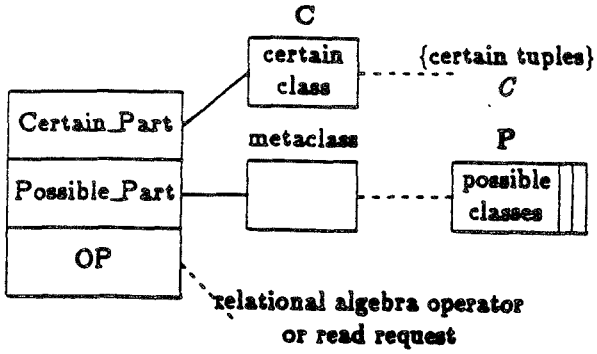


Figure 5(a). Approximate class

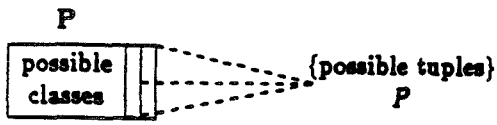


Figure 5(b). Possible classes

Figure 2 and Figure 4. Figure 6(a) shows an approximate object $\mathbf{R}_3$ corresponding to the relation R_3 in Figure 2(d). The set C_3 of certain tuples in $R_3 = (C_3, P_3)$ are all the instances of the classes, **Urbana Ford 4–door sedans and Champaign Ford 4–door sedans, that have the color red.** The possible tuples in P_3 are instances of $\mathbf{P}_3$, which is **{Springfield Ford 4–door sedans}.**

Given a set of approximate objects corresponding to all approximate relations of a standard relation S, we define a partial order $\geq$ over the set as follows. The approximate object $\mathbf{R}_i$ is said to be better than or equal to the approximate object $\mathbf{R}_j$, denoted as $\mathbf{R}_i \geq \mathbf{R}_j$, if (1) the certain set of $\mathbf{R}_i$ contains the certain set of $\mathbf{R}_j$, (2) the set of possible classes of $\mathbf{R}_i$ are a subset of the set of possible classes of $\mathbf{R}_j$, and (3) $C_i - C_j$ is a subset of the instances of the possible classes in $\mathbf{P}_j - \mathbf{P}_i$. In other words, $C_i \supseteq C_j$ and $\mathbf{P}_i \subseteq \mathbf{P}_j$. Let R_i and R_j denote two approximate relations corresponding to the two approximate objects $\mathbf{R}_i$ and $\mathbf{R}_j$, respectively. If $\mathbf{R}_i \geq \mathbf{R}_j$, then the corresponding approximate relations are such that $R_i \geq R_j$.

3.3. Methods of Approximate Classes

An approximate class definition contains five methods: new, migration, specialisation, generalisation, and instantiation. These methods are invoked during query processing. The new method causes an approximate object to be created and initialised. The specialisation method uses information contained in the class hierarchy to determine the subclasses of a specified class. The generalisation method uses information contained in the class hierarchy to determine the class(es) of a set of instances. The migration method results in an improved approximation, whereby a subset of all possible instances of one or more class in P migrates to the certain set C and the corresponding class(es) in P is removed from P. The instantiation method causes the instances of a set of possible classes of an approximate object to be generated.

Car No.	Location
5421	Urbana
9021	Champaign
9122	Champaign
{Springfield Ford 4–door Sedans}	

Figure 6(a). $\mathbf{R}_3$: An approximate object.

Car No.	Location
5421	Urbana
7824	Springfield
7924	Springfield
{Champaign Ford sedans color=red}	

Figure 6(b). $\mathbf{R}_5$: An approximate object

In the following, we first define the migration, specialisation, and generalisation methods informally in terms of their effects on the value of an approximate object. Their use in query processing is described in the next section. Let γ be a method, its arguments and result being approximate objects. Let $\mathbf{R}_1$ and $\mathbf{R}_2$ be two approximate objects and values of an argument to γ. A method is said to be monotone if $\mathbf{R}_1 \geq \mathbf{R}_2$ implies that $\gamma(\mathbf{R}_1) \geq \gamma(\mathbf{R}_2)$ when the other arguments to γ are equal. The proof of the monotonicity of the generalisation method appears in Appendix B, the remaining proofs in [14].

Let $\mathbf{L}^I$ denote the set of all instances of a class L. Given a set of classes P, let $\mathbf{P}^I = \bigcup_{\mathbf{L} \in \mathbf{P}} \mathbf{L}^I$. In other words, $\mathbf{P}^I$ denotes the union of the sets of all instances of the classes in P. Given a set P of possible classes in an approximate object, $\mathbf{P}^I = P$ is the set of possible tuples in the corresponding approximate relation.

The migration method has as its arguments a set of certain instances C_i and a set of possible classes $\mathbf{P}_i$. We denote the migration of an approximate object by migr $(\mathbf{C}_i, \mathbf{P}_i)$. The migration method results in an improved approximation, whereby a subset of all possible instances of one or more class in

P_i migrates to the certain set C_i and the corresponding class(es) in P_i is removed from P_i. The migration method is invoked during query processing. We will return to describe it in more detail in Section 4.

The specialisation method has as an argument a class L in the set of possible classes P. The specialisation of class L is indicated by spec(L, q) where q is a qualification. The specialisation method uses information contained in the class hierarchy. Specifically, the subclasses of the class L are determined, and a subset of these subclasses that match the qualification q is selected. spec(L, q) causes the value of the set P to be updated so that the class L is replaced by the selected subset of its subclasses. spec(L) indicates that all the subclasses of L are to be selected. The specialisation method satisfies the following relationships: $(\text{spec}(L, q))^I \subseteq L^I$ and $(\text{spec}(L))^I = L^I$.

The generalisation method has as an argument a set of instances L. We denote the generalisation method by genl(L). generalisation uses information contained in the class hierarchy to determine the class(es) of a set of instances. The generalisation method does not change the value of the approximate object; it only determines the class(es) of its argument. The generalisation method satisfies the following relationships: $(\text{genl}(L))^I \supseteq L^I$, $\text{genl}(L^I) = L$, and $(\text{genl}(L^I))^I = L^I$.

Corresponding to the set of approximate relational algebra operations in Table 1 that operate on approximate relations, we have a set of approximate relational algebra operations that operate on the corresponding approximate objects. These operations are defined in terms of the generalisation method, and their definitions are listed in Table 2 of Appendix A. Again, R_1 and R_2 are approximate objects, and $R_1 = (C_1, P_1)$ and $R_2 = (C_2, P_2)$. In Table 2, $\times$, $\cup$, $\cap$, and $-$ in the C_T and P_T columns denote the set-theoretical operations cartesian product, union, intersection, and set difference, respectively.

During query processing in the approximate relational data model, the possible tuples are generated, and relational algebra operations are applied to them. In contrast, the object-oriented query processor maintains a template of the possible tuples as classes in P, in order to avoid performing repeated relational algebra operations on the actual possible tuples every time an update of the value of a leaf node is propagated up the query tree. The information contained in the possible classes provides an alternative to the possible tuples. As shown in Table 2, the relational algebra operations are performed on the certain instances and on the possible classes.

4. Object-Oriented Query Processing

After a query is parsed, the first step of query processing is to create an approximate class for each node in the query tree. The schema of the relation represented by a node in the query tree can be determined from the schemas of the base relations in the subtree rooted at this node, and the relational algebra operations at and below this node. This schema determines the attributes in the certain class C of the approximate class that defines all approximate relations of this relation. The query processor makes use of the information on the schemas of all base relations provided by the class hierarchy to accomplish this task.

After an approximate class is created for each node, the new method of each approximate class is used to create an approximate object of that class. An approximate object is treated as a temporary object which only exists during the processing of the query. A newly created approximate object initialises its instance variable OP to the relational algebra operation whose result is the value of the node. In the case of a leaf node, the value of OP is read_request. The set of instances of the certain class of every approximate object is initially empty; the corresponding approximate relation contains an empty subset of certain tuples.

The appropriate set of possible classes for the new approximate object is determined based on information provided by the class hierarchy. For example, consider the query: "Select all red, Ford sedans." One initial value of P, the set of possible classes of the approximate object at the root node, is {cars}, which contains the requested domains. While this is a valid choice, leading to a valid approximate answer, this class may have a large number of instances that are not members of the certain set, such as, instances that belong to the class Chrysler. Based on information in the class hierarchy shown in Figure 4, the specialisation method can be invoked to provide a better approximation than the one given by (Ø, {cars}), such as (Ø, {Ford sedans}).

The specialization method is also invoked to determine the most specialized class for the set of possible classes of each leaf node. If a base relation is indexed on an attribute which appears in the select clause of a query, then there exists a set of subclasses in the class hierarchy specialized on that attribute. The use of one of these specialized subclasses in the set of possible classes of a leaf node takes the place of the corresponding select clause when processing the query. In the above example, the P of one of the leaf nodes is initialized to {**Ford sedans**}, replacing the select on Ford and the select on sedans. Only the select on the color red is needed when processing the query.

During query processing, **read_request's** are issued by the approximate objects at the leaves of the query tree. A **read_request** to retrieve a base relation will return all database objects in the corresponding class unless some segments of the relation are inaccessible during the time allocated by the query processor to carry out the **read_request.** When some segments of the relation are not accessible, a partial read is executed, and the accessible part of the relation is read and returned. Therefore, the result of a **read_request** is either all instances of the certain class C (of the approximate object at the leaf node) or a subset of the instances of the class C. In the latter case, a **read_request** to a relation is effectively a sequence of partial reads, each of which returns a segment. Improvements in the query result can be propagated up the query tree each time a partial read is complete, after several partial reads, or after all partial reads have been completed. In any case, when the result of a **read_request** is returned, a subset of the instances of the certain class C is said to be instantiated. Hereafter, by instantiation of C we mean a subset of instances of C is instantiated at a time, and the instances in this subset are database objects returned by the corresponding **read_request.**

The propagation mechanism involves propagating improvements in the quality of the approximations upward in the query tree, and it is implemented by the migration method. The values of C and P of an approximate object at a leaf node are updated when database objects are returned by the **read_request** to retrieve the base relation. These objects are treated as instances of the class C, and the corresponding class(es) in P is deleted. The new values of C and P are sent to the approximate object at the parent node of the leaf node. The migration method is invoked when an approximate object receives a new set of values of an approximate object at a node that is its child. It recomputes its own value by applying its relational algebra operator OP to the newly received arguments. This resulting value is then sent upward to the approximate object at its parent node in the query tree. While query processing progresses, as more database objects are returned, improved approximations of the answer are created in which the set of certain instances grows and the set of possible classes shrinks. The propagation mechanism causes some possible instances to become members of the certain set and some to be eliminated, and the corresponding possible classes to be deleted from P.

The classes in P can be classes from the class hierarchy. These are called *base* classes. When approximate relational algebra operations such as select and project are applied to possible classes, new classes are created which may not be in the class hierarchy. These new classes are called *derived* possible classes. For example, an approximate object R_5 corresponding to an approximate relation on Ford, red sedans appears in Figure 6(b). The corresponding approximate relation R_5 is shown in Figure 2(f). The set C_5 of certain tuples are instances of the classes, **Urbana Ford 4–door sedans** and **Springfield Ford 4–door sedans**, that are red. The possible class in P, **Champaign Ford sedans** color=red, has as its instances the possible tuples on Ford, red sedans in Champaign. This class is derived from the class, **Champaign Ford sedans**, in the class hierarchy when the select operator $\sigma_{color=red}$ is applied to it.

If time permits for the completion of a query, the set of certain instances C_i in the approximate object (C_i, P_i) at the root node is the set of tuples S, which is the exact answer to the query, and the set of possible classes in P_i is empty. If query processing must be terminated before it is completed, the approximate answer at this time consists of a set of certain instances, which is a subset of the exact answer S, and a set of possible classes whose instances are tuples possibly in the answer. An instantiation method is provided for a possible class. The query processor can invoke this method to generate the set of all instances of every possible class remaining in P_i. This method computes all the tuples in the cartesian product of the domains in the relation corresponding to the possible class. This can be done without reading any data. In some cases, the class

descriptions of the possible classes may provide the desired information on possible tuples to the the user. The possible tuples are not generated, only the class descriptions of the possible classes are returned along with the set C_i as part of the approximate answer.

5. Conclusions and Future Explorations

The object–oriented query processor described here provides a means to implement the approximate relational model and monotone query processing efficiently. Specifically, it provides a way to identify the initial approximation of the answer. Query processing overhead is reduced from that encountered without the aid of the object–oriented model because the methods used to implement the approximate relational algebra operations are not performed on the possible tuples but on the possible classes. In contrast, in the case of the query processing algorithm based solely on the approximate relational algebra model, relational algebra operations are performed on both the certain tuples and the possible tuples each time improvements in the approximations of the base relations are propagated up the query tree. The additional overhead required using the object–oriented model is that of maintaining and retrieving information in the class hierarchy, and we are currently examining this overhead. Again, the object–oriented query processor requires only minimal changes be made to the underlying relational database system in order to support the view represented by the class hierarchy. The object–oriented query processor can be one of several query processors available to the database system. We are currently implementing such a query processor as well as exploring the use of other types of class hierarchies to make query processing more efficient.

Our current work has demonstrated that the object–oriented model described here can be extended to support approximate query processing where a broader notion of approximations may be adopted. We are examining the use of a class lattice with multiple inheritance in order to represent different semantic relationships between database objects. We are including features to determine how close an approximate answer is to the exact answer according to the new semantics.

References

[1] Lin, K. J., S. Natarajan, J. W. S. Liu, and T. Krauskopf, "Concord: A System of Imprecise Computations," *Proc. COMPSAC '87,* Tokyo, Japan, pp. 75–81, Oct. 1987.

[2] Chung, J. Y., J. W. S. Liu, and K. J. Lin, "Scheduling Periodic Jobs that Allow Imprecise Results," *IEEE Trans. on Computers,* Aug. 1990.

[3] Smith, Kenneth P., and J. W. S. Liu, "Monotonically Improving Approximate Answers to Relational Algebra Queries," *Proc. COMPSAC '89,* Orlando, Florida, Sept. 1989.

[4] Tsvieli, Arie, "Representation and Access of Uncertain Relational Data," *Data Engineering,* Vol. 12, No. 2, pp. 21–28, June 1989.

[5] Liu, K. and R. Sunderraman, "On Representing Indefinite and Maybe Information In Relational Databases," *Proceedings: 4th Conf. on Data Engineering,* pp. 250–257, Feb. 1988.

[6] Imielinski, Tomass, "Incomplete Information in Logical Databases," *Data Engineering,* Vol. 12, No. 2, pp. 29–40, June 1989.

[7] Zemankova, Maria, "FIIS: A Fuzzy Intelligent Information System," *Data Engineering,* Vol. 12, No. 2, pp. 11–20, June 1989.

[8] Motro, Amihai, "VAGUE: A User Interface to Relational Databases that Permits Vague Queries," *ACM Trans. on Office Info. Sys.,* Vol. 6, No. 3, pp. 187–214, July 1988.

[9] Davidson, S. B., and A. Watters, "Partial Computation in Real–Time Database Systems," *Proc. 5th Workshop on Real–Time Software and Operating Systems,* pp. 117–121, May 1988.

[10] Buneman, P., S. Davidson, and A. Watters, "A Semantics for Complex Objects and Approximate Queries," *Proc. 7th Symposium on the Principles of Database Systems,* pp. 305–314, March 1988.

[11] Hull, Richard, and Roger King, "Semantic Database Modeling: Survey, Applications, and Research Issues," *ACM Computing Surveys,* Vol. 19, No. 3, pp. 201–260, September 1987.

[12] Kim, Won, et. al., "Integrating an Object–Oriented Programming System With a Database System," *OOPSLA Proceedings,* pp. 142–152, Sept. 1988.

[13] Goldberg, Adele, and David Robson, *Smalltalk–80: The Language and Its Implementation,* Addison–Wesley, MA, 1983.

[14] Vrbsky, S. V., Jane W. S. Liu, "An Object–Oriented Query Processor," *TR No. UIUCDCS–R–90–1706,* Dept. of CS, UIUC, Feb. 1990.

Appendix A

Approximate Operation	C_T	P_T
Union: $R_T = R_1 \cup R_2$	$C_T = C_1 \cup C_2$	$P_T = (P_1 \cup P_2) - C_T$
Difference: $R_T = R_1 - R_2$	$C_T = C_1 - R_2$	$P_T = (P_1 - R_2) \cup (P_2 \cap R_1)$
Select: $R_T = \sigma_{att=val} R_1$	$C_T = \sigma_{att=val} C_1$	$P_T = \sigma_{att=val} P_1$
Project: $R_T = \pi_{att} R_1$	$C_T = \pi_{att} C_1$	$P_T = \pi_{att} P_1$
Cart. Prod: $R_T = R_1 \times R_2$	$C_T = C_1 \times C_2$	$P_T = (R_1 \times R_2) - C_T$

Table 1: Approximate relational operations

Approximate Operation	C_T	$\mathbf{P}_T$
Union: $\mathbf{R}_T = \mathbf{R}_1 \cup \mathbf{R}_2$	$C_T = C_1 \cup C_2$	$\mathbf{P}_T = (\mathbf{P}_1 \cup \mathbf{P}_2) - \mathbf{genl}(C_T)$
Difference: $\mathbf{R}_T = \mathbf{R}_1 - \mathbf{R}_2$	$C_T = C_1 - C_2 - \{ t \mid \mathbf{genl}(t) \in \mathbf{P}_2 \}$	$\mathbf{P}_T = (\mathbf{P}_1 - \mathbf{genl}(C_2) - \mathbf{P}_2) \cup (\mathbf{P}_2 \cap \mathbf{genl}(C_1) \cap \mathbf{P}_1)$
Select: $\mathbf{R}_T = \sigma_{att=val} \mathbf{R}_1$	$C_T = \sigma_{att=val} C_1$	$\mathbf{P}_T = \sigma_{att=val} \mathbf{P}_1$
Project: $\mathbf{R}_T = \pi_{att} \mathbf{R}_1$	$C_T = \pi_{att} C_1$	$\mathbf{P}_T = \pi_{att} \mathbf{P}_1$
Cart. Prod: $\mathbf{R}_T = \mathbf{R}_1 \times \mathbf{R}_2$	$C_T = C_1 \times C_2$	$\mathbf{P}_T = ((\mathbf{genl}(C_1) \times \mathbf{P}_1) \cup (\mathbf{genl}(C_1) \times \mathbf{P}_2) \cup (\mathbf{genl}(C_2) \times \mathbf{P}_1) \cup (\mathbf{genl}(C_2) \times \mathbf{P}_2)) - \mathbf{genl}(C_T)$

Table 2: Approximate relational algebra operations using approximate objects

Appendix B

Theorem: The generalisation method is monotone.

Proof: We define the generalisation $\mathbf{R}_T = (C_T, \mathbf{P}_T)$ of an approximate object, $\mathbf{R}_1 = (C_1, \mathbf{P}_1)$, as follows:

$$C_T = C_1$$
$$\mathbf{P}_T = \mathbf{genl}(\mathbf{P}_1)$$

In other words, the set of possible classes of $\mathbf{R}_T$ is the set of superclasses of the classes in $\mathbf{P}_1$. To show that generalization is monotone, let $\mathbf{R}_1$, $\mathbf{R}_1'$, $\mathbf{R}_T$, and $\mathbf{R}_T'$ be approximate objects. Let $\mathbf{R}_1' \geq \mathbf{R}_1$, and therefore the corresponding approximate relations are such that $R_1' \geq R_1$. Suppose that $\mathbf{R}_T = (C_1, \mathbf{genl}(\mathbf{P}_1))$ and $\mathbf{R}_T' = (C_1', \mathbf{genl}(\mathbf{P}_1'))$. Let P_1, P_1', P_T, and P_T' be the sets of instances of the classes in $\mathbf{P}_1$, $\mathbf{P}_1'$, $\mathbf{P}_T$, and $\mathbf{P}_T'$ respectively. To show that $\mathbf{R}_T' \geq \mathbf{R}_T$, we need to show that (1) $C_T' \supseteq C_T$ and (2) $P_T' \subseteq P_T$.

To show that $C_T' \supseteq C_T$, we consider an arbitrary instance t in C_1. Clearly, $t \in C_1'$ since $C_1' \supseteq C_1$. By definition, if $t \in C_1$ then $t \in C_T$ since $C_T = C_1$. An instance in C_T is also always in C_T' since $C_T' = C_1'$. Hence, $C_T' \supseteq C_T$. To show that $P_T' \subseteq P_T$, we consider an arbitrary instance t in P_1'. If $t \in P_1'$ then $t \in P_1$ because $P_1' \subseteq P_1$. Therefore, there does not exist a t such that $t \in P_1'$ and $t \notin P_1$. By definition a superclass is partitioned by its subclasses. Therefore, if $\mathbf{genl}(\mathbf{P}_1') \supseteq \mathbf{genl}(\mathbf{P}_1)$, then there exists a t such that $t \in P_1'$ and $t \notin P_1$, which contradicts $P_1' \subseteq P_1$. Therefore, $\mathbf{genl}(\mathbf{P}_1') \subseteq \mathbf{genl}(\mathbf{P}_1)$. Since $\mathbf{genl}(\mathbf{P}_1') = \mathbf{P}_T'$ and $\mathbf{genl}(\mathbf{P}_1) = \mathbf{P}_T$, it follows that an arbitrary instance in P_T' is always also in P_T, and $P_T' \subseteq P_T$. A similar proof can be given for the generalisation method applied to a set of instances. ■

S3-61
93852
245134
15 p.

PERTS: A Prototyping Environment for Real-Time Systems

Jane W. S. Liu, Kwei-Jay Lin and C. L. Liu
Department of Computer Science
University of Illinois

ABSTRACT

We discuss an ongoing project to build a Prototyping Environment for Real-Time Systems, called PERTS. PERTS is a unique prototyping environment in that it has (1) tools and performance models for the analysis and evaluation of real-time prototype systems, (2) building blocks for flexible real-time programs and the support system software, (3) basic building blocks of distributed and intelligent real-time applications, and (4) an execution environment. PERTS will make the recent and future theoretical advances in real-time system design and engineering readily usable to practitioners. In particular, it will provide an environment for the use and evaluation of new design approaches, for experimentation with alternative system building blocks, and for the analysis and performance profiling of prototype real-time systems.

I. INTRODUCTION

In recent years, a great deal of progress has been made in software engineering technology. There are now good commercially available and experimental prototyping and software engineering environments that target all software development phases. While the existing and emerging approaches are effective for most application domains, they often do not address the difficulties in building hard real-time computing systems. A hard real-time computing system is one in which there is a *timing constraint* imposed on every time-critical task. The simplest timing constraint is given as a *deadline*. For the result produced by a time-critical task to be correct, it must be available by the task's deadline, in addition to being functionally correct. Specific applications supported by real-time systems include command and control, guidance and navigation, flight control, vision and object identification, autonomous vehicle control, robots, process control, and intelligent manufacturing.

The approach that has been taken traditionally to construct real-time systems is to develop the application software first and then tune the application and the underlying support system to make sure that all the timing constraints are met. The ad hoc and time-consuming methods used in the latter step tend to produce brittle, difficult-to-modify and hard-to-maintain real-time systems. Small changes in the application software, or in the underlying hardware and software support, can produce unpredictable, devastating timing effects that can be detected and corrected only through exhaustive testing and performance tuning. Consequently, it is costly to develop new systems and to upgrade, extend or port existing systems. Rapid advances in hardware and software technologies have greatly accelerated the capabilities of, and in turn have increased the demands for, real-time systems. As the applications of real-time systems become more critical and the underlying systems become more complex, the traditional methods can no longer be relied on for building provably responsive and robust systems.

The lack of effective methods and tools for building future real-time systems has motivated the recent research on the theoretical foundations of real-time computing. One goal of the research is to find

In *Proceedings of 1991 System Design Synthesis Technology Workshop*, Silver Spring, MD, September 1991.

methods for predicting the timing behavior of the basic building blocks and the overall real-time systems synthesized from the basic building blocks. Another goal is to develop integrated approaches to building real-time systems, as alternatives to the traditional approach. Rather than designing the time-critical applications and the scheduling and controlling mechanisms for them independently, an integrated approach would begin with models and optimality criteria that explicitly account for the possibility of trading off between various figures of merit, and then design the application and the underlying system to achieve the desired tradeoffs. Such an approach would lead to more flexible, easy-to-schedule programs, and resultant systems would degrade gracefully during overloads and failures.

This paper describes an ongoing project on building a real-time programming environment at the University of Illinois. This environment is called PERTS (Prototyping Environment for Real-Time Systems). PERTS is a unique in that it has

(1) tools and performance models for the analysis and evaluation of prototype real-time systems, — The PERTS tools are capable of providing worst-case bounds and performance predictions needed to ensure the responsiveness and robustness of time-critical applications. Some of these tools were made feasible only recently by new theoretical advances in real-time scheduling.

(2) building blocks for the implementation of flexible real-time programs and the system software, — These system components are based on the imprecise computation approach and will support the use of, and experimentation with, this new technique.

(3) basic building blocks of the underlying support system for distributed and intelligent real-time applications, — These reusable building blocks will realize existing and new scheduling algorithms and resource access control protocols and make them ready for use in future systems.

(4) an execution environment. — This environment will allow experimental evaluation of the alternatives in scheduling the target software system.

The rest of the paper is organized as follows. Section II provides a review of some of our past work. The capabilities of PERTS and its key components are presented in Section III. Section IV discusses the PERTS programming language and the execution environment. We compare this project with other related projects in Section V. Section VI is a summary.

II. PREVIOUS WORK

We have been working on many problems related to the design and implementation of real-time systems, including: (1) design and evaluation of algorithms and protocols, (2) schedulability analysis of real-time programs, (3) measurement-based analysis of program execution times, (4) characterization of benchmark real-time workloads, and (5) real-time programming environments. This section summarizes our past work that is relevant to PERTS.

II.1. Scheduling Algorithms and Protocols

Many real-time applications, such as flight control and tracking, can be characterized by the well-known periodic-job model. The workload on such a system consists of periodic jobs, each job being a periodic sequence of requests for execution of the same computation (for example, a control law or a

digital filter.) The time between the arrivals of two consecutive requests in a job is called its period. Every request has a deadline; it must be completed at the end of its period of arrival. The fraction of time a set of jobs keeps a processor busy is called its total utilization.

A well-developed, rigorous method for the design and synthesis of real-time systems for applications characterizable by the periodic-job model is based on the rate-monotone paradigm [1,2]. According to this paradigm, jobs in a multiprocessor system are assigned to processors statically, and jobs on each processor are scheduled using the rate-monotone algorithm. We have developed an effective scheme [2] for assigning periodic jobs to processors. The rate-monotone algorithm is preemptive and priority-driven. It assigns fixed priorities to jobs on the basis of their periods; the shorter the period of a job, the higher its priority. We have shown [1] that as long as the total utilization of a set of K jobs is less than or equal to $K(2^{1/K}-1)$ (that is, 0.82 for $K=2$ and ln 2 for large K), no deadline will be missed when the jobs are scheduled using this algorithm. Often, job sets with total utilization larger than this worst-case bound can also be feasibly scheduled, meeting all deadlines, using this algorithm. We have also shown that in a rate-monotone schedule, all deadlines are met as long as the first deadline of every job is met. This property, along with the above mentioned worst-case bound, eliminates the need for exhaustive tests to verify that all deadlines are met at all times. Consequently, a multiprogrammed real-time system based on the rate-monotone paradigm is relatively easy to modify and maintain. Many researchers have enhanced the rate-monotone algorithm to handle more complex periodic jobs, such as jobs that share resources, or are mixed with aperiodic jobs, and so on. Recently, we have extended this algorithm to schedule periodic jobs that have deferred deadlines [3]. Our preliminary results show that the extended algorithm, called the modified rate-monotone algorithm, offers an effective way to schedule heterogeneous mixes of jobs with different timing requirements.

II.2. Real-Time Computations Models

We have proposed the imprecise computation technique [4-18], called the anytime or sufficiently-good paradigm in AI literature, as a way to cope with the difficulty in meeting all deadlines at all times in dynamic systems. This technique prevents timing faults and achieves graceful degradation by making an approximate result of an acceptable quality available in time when the result of the desired quality cannot be produced in time. In a system that supports imprecise computations, every time-critical task is structured in such a way that it can be logically decomposed into a mandatory subtask and an optional subtask. The mandatory subtask is the portion of the computation that must be done before the deadline of the task in order to produce a result of an acceptable quality. The optional subtask is the portion of the computation that refines the result and can be left unfinished if necessary at the expense of the quality of the result produced by the task. The imprecise computation technique makes meeting all deadlines and, at the same time, making near-optimal use of system resources significantly easier because we only need to ensure that all the mandatory subtasks are completed by their deadlines. The leftover system resources can be used to complete as many optional subtasks as possible. Only the mandatory subtasks are restricted to have bounded execution time and resource requirements. It is not necessary to eliminate non-determinism in the timing requirements of optional subtasks.

We have developed several workload models to characterize imprecise computations [19-28]. These models explicitly quantify the costs and benefits in the tradeoff between the result quality and

computation time requirements. For different types of applications, the costs and benefits are more appropriately measured by different criteria. Consequently, there are many different imprecise scheduling problems. Many scheduling algorithms have been developed to solve these problems, including algorithms for scheduling imprecise jobs on uniprocessors, jobs with multiple versions, and parallelizable and multitasking computations on multiprocessor systems. The imprecise computation technique also relies on the use of well-behaved computational algorithms that produce acceptable, intermediate results in predictable, short amounts of time and better results when allowed more time. We have developed computational algorithms with the desired characteristics in some of the application domains where such algorithms did not exist [29,30].

We have also developed other new workload models to characterize real-time applications that cannot be adequately characterized by the traditional models. One of these models is the AND/OR model [31]. In the traditional model, if a task has more than one immediate predecessor, then its execution cannot begin until all of its immediate predecessors have completed. In the AND/OR model, such a task is called an AND task. The execution of an OR task with more than one immediate predecessor can begin as soon as one of these predecessors has completed. This model has three variants. In some applications, all predecessors of an OR task must eventually be completed. In other applications, some immediate predecessors of an OR task may be skipped entirely. In a third type of application, OR predecessors are executed in turn until a "success" occurs, and the scheduler employs knowledge of the probability of success for each predecessor. This variant of the AND/OR model can be used to characterize applications such as a heuristic search where disjoint search spaces are examined concurrently and the subsequent step, the OR task, can begin execution as soon as the search in any one of the spaces is successful. We have developed polynomial-time optimal algorithms for scheduling to meet deadlines in special cases where the AND/OR scheduling problem is not NP-hard and have evaluated heuristic algorithms that can be used to produce sub-optimal schedules in the other cases.

In the traditional models, deadlines of tasks are static and are independent of the way tasks are scheduled and executed. The distance-constraint model [32] characterizes tasks that have dynamic deadlines. Requiring that the executions of the tasks meet dynamic deadlines is the same as requiring their executions be within some distances in time from each other. This model occurs in the scheduling of transactions in reactive databases and acknowledgements in reliable networks. We have been concerned with ways to schedule distance-constrained tasks on a single processor and have developed polynomial-time optimal algorithms for several special cases of practical interest.

The future real-time systems are likely to be implemented on distributed and parallel hardware. Flow shops and job shops model such systems in a natural way. When tasks execute on different processors in turn, the problem of end-to-end scheduling on the processors is a job-shop problem, if the order in which tasks execute on different processors is arbitrary, or a flow-shop problem, if the order is identical. We have extended the traditional flow-shop model to two more variants: flow shop with recurrence and periodic flow shop [33]. We have designed and evaluated a simple heuristic algorithm for scheduling tasks with arbitrary processing times in traditional flow shops and a polynomial-time optimal algorithm for scheduling homogeneous tasks in simple flow shops with recurrence. We are currently evaluating several heuristic algorithms, each of which is designed to near-optimally schedule a different job mix.

III. SYSTEM DESIGN USING PERTS

One of the goals in the PERTS project is to implement, maintain, and distribute an extensible library of scheduling algorithms and resource access protocols that are the basic building blocks of complex and intelligent real-time systems. PERTS will also provide an integrated set of analysis and simulation tools, together with an execution environment, in order to evaluate each of these basic building blocks as well as the complete systems built from them. In particular PERTS will provide a set of building blocks for prototyping real-time systems based on the imprecise computation approach [4-18]. The PERTS tools and building blocks will be built in such a way that they can be easily integrated into other object-based prototyping and software engineering environments designed for specific application domains and operating systems.

The remainder of this section describes the capabilities of PERTS and its key components. Throughout this paper, we refer to each granule of computation or communication that is to be allocated processor time and other system resources as a *task*. A software system, also called a *task system*, is a set of tasks which are dependent on each other either because there are data and functional dependencies between them or because they share some resource.

III.1. Capabilities

In Figure 1, we show the components of PERTS. PERTS has two sets of components. The first set is unique to PERTS, including the Description Library, the Schedulability Analysis System and the Testbed. The second set of the components include those tools which exist in many other real-time prototyping environments, such as an intelligent compiler, timing analysis tools, and a workload generator.

PERTS can be used to support the design of real-time systems. The design of a target task system is captured by its abstract description provided by the user. We will return in Section IV to discuss the abstract description. At the abstract level, it is sufficient to characterize the underlying systems in terms of the amounts of computational, storage, input/output, and communication resources available to support the target task system. The description of each resource provides the constraints governing its usage, such as whether it is shared, etc. During the design phase, the schedulability analysis system will serve as an interactive tool. This tool can be used for many purposes, including to determine whether sufficient amounts of all resources are available; to identify potential bottlenecks; to select computational algorithms from available choices with different levels of result quality versus resource requirements; and to provide suggestions on the choices of task parameters, such as the periods of the periodic tasks, granularities of distributed modules, and alternative dependency relationships between tasks. The analysis tool and performance predictor can be used to identify where later changes in software or hardware are likely to lead to unpredictable timing effects. In this way, the schedulability analysis system can also help in the design of any test suite which will be needed later to test the target system.

PERTS will provide similar support in later phases of software development. It can generate from annotated source code a concrete description of the target software system. In an early development stage, PERTS can be used to identify and choose a set of operating system policies for task partitioning and assignment, load balancing, scheduling and resource management. In this case, the concrete

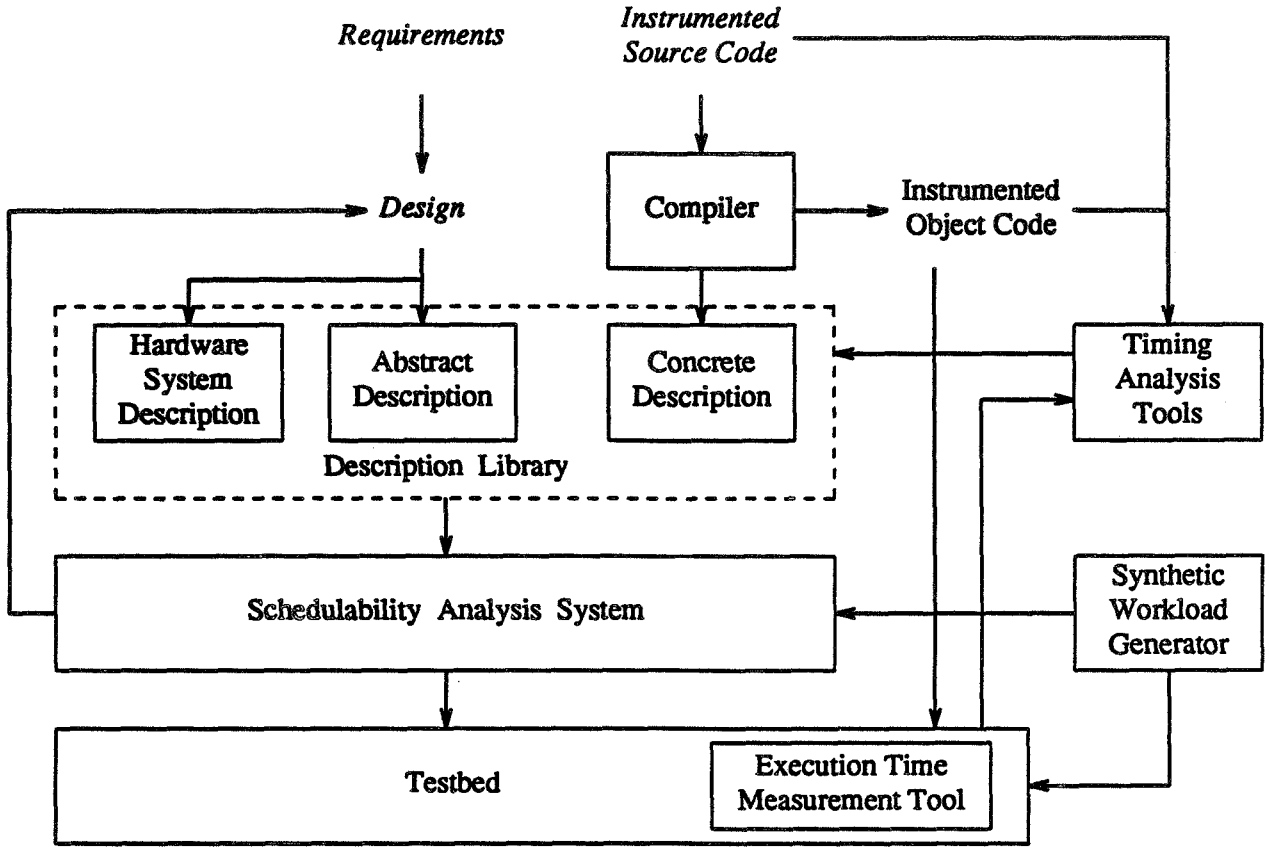


Figure 1. The Prototyping Environment of Real-Time Systems

description may simply be a more detailed data-flow graph that contains sufficiently accurate information about the timing and resource usage characteristics of the tasks. PERTS will produce sample task assignments, schedules, memory layouts, etc. to provide the feedback needed in the iterative software development process. In later stages, when the source code of the target task system becomes available, PERTS can provide an execution environment in which the target system can be debugged to a great extent and its performance profile measured. In this case, the concrete description is the instrumented object code generated by a cross language compiler. This code can be run, under the scheduling directives produced by the scheduling analysis system, in a simulated or emulated target environment provided by the testbed The testbed will contain a workload generator capable of generating synthetic or trace-driven workloads to support the emulation and simulation of the embedded environment.

An essential class of tools that will be provided in PERTS consists of program execution time analysis and measurement tools. In addition to a tool that determines the maximum execution times of tasks by examining and analyzing source and object codes of the programs, a measurement-based timing analysis tool [16,18] will also be provided to determine the execution times of dynamic programs and programs in parallel and distributed environments. We also plan to maintain a library of locality and

timing behavior models of commonly used real-time programs obtained from traces of their executions.

III.2. Schedulability Analysis System

The key component of PERTS is the schedulability analysis system shown in Figure 2. The unique features of this system include

(1) an extensible set of optimal, approximate, and heuristic algorithms to deal with the difficult problems of scheduling tasks to meet timing constraints in high-performance, intelligent, distributed and parallel systems, and

(2) a comprehensive set of analysis, simulation and measurement tools capable of providing rigorous worst-case bounds and complete performance profiles to support the use of these algorithms.

The schedulability analysis system will produce timing diagrams, memory maps, load conditions, etc. when used as an interactive design tool. It can also be used to simulate the schedulers and protocol

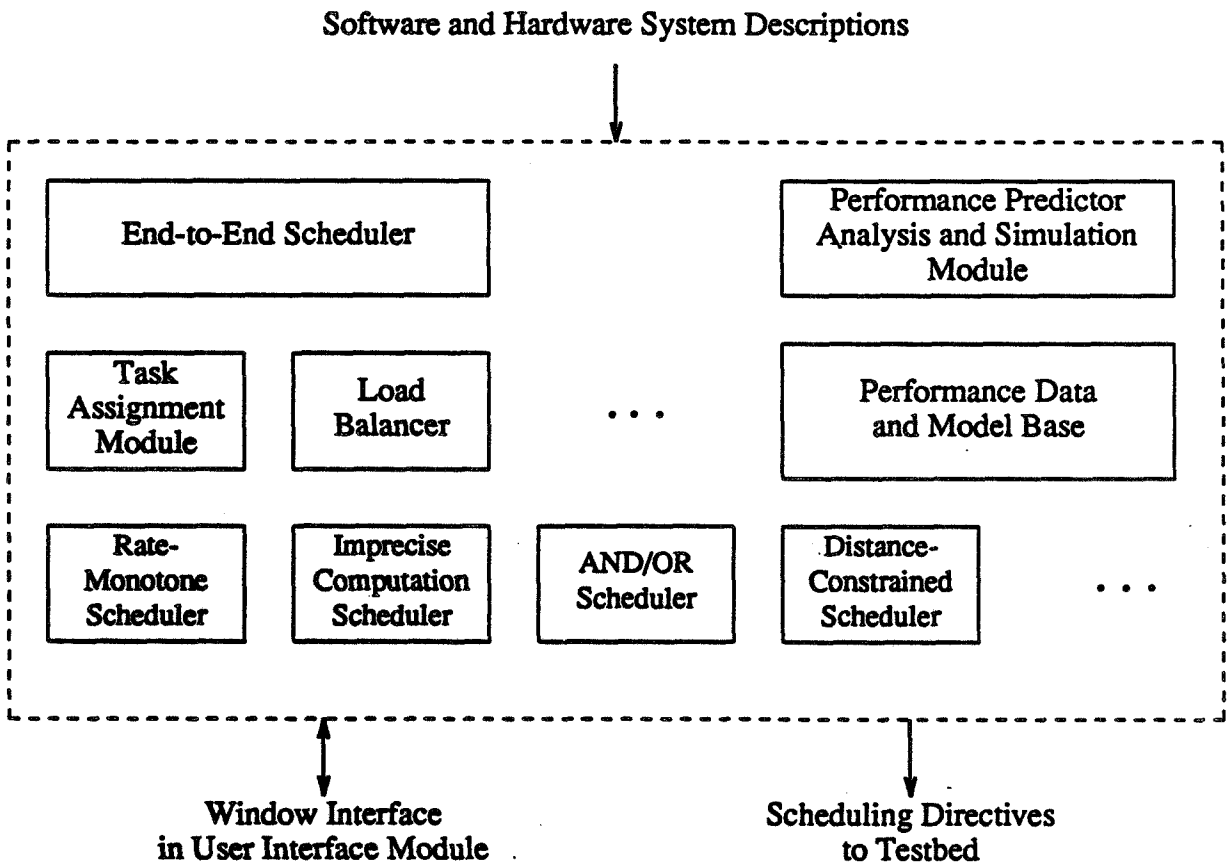


Figure 2. Schedulability Analysis System

handlers in the target operating system when one wants to run the target task system on the testbed, scheduling and dispatching the tasks and controlling resource accesses during task executions. The schedulers and protocol handlers in PERTS will be implemented in C++. Each of them can be customized and used as a building block of the target operating system. To make the selection of the algorithms and the use of the tools in PERTS as easy as possible, a browsing and help facility will be provided.

The schedulability analysis system will support the hierarchical approach to building large and complex, real-time software on distributed and parallel hardware platforms. Examples of algorithms that will be implemented for this purpose include algorithms for end-to-end scheduling of distributed tasks that have overall deadlines; algorithms for scheduling parallelizable tasks with deadlines on massively parallel systems; partitioning and assignment schemes for statically assigning tasks to processors; load balancing algorithms for dynamic adjustment of load conditions; and protocols for controlling concurrent access to resources and data transmissions. For example, the task partitioning and assignment module can help the designer to find a partition and assignment of the given task system so that the tasks assigned to each processor can meet their individual deadlines and the overall task system can meet its end-to-end deadlines. When the target system described by the given abstract description does not have such an assignment, the analysis tool in the system can suggest possible changes to make such an assignment feasible. If a dynamic task assignment approach is chosen for any reason, the performance predicting tool can be used to determine whether the worst-case performance of the assignment is acceptable.

Like many experimental and operational scheduling tools, this system will support most commonly used scheduling disciplines, such as the rate-monotone discipline [1], the first-in-first-out discipline, priority-driven disciplines, etc. It will differ from the existing systems in that it will also provide schedulers built on several new workload models; these new models allow us to characterize many real-time applications that cannot be adequately characterized by traditional models. Examples include the models described in Section II: AND/OR precedence-constrained task systems that are encountered in applications as diverse as distributed control systems, robotic assembly, and concurrent heuristic searches [31]; multiple-version tasks used to enhance fault tolerance of real-time systems; distance-constrained tasks that are found in reactive systems such as real-time monitors [32]; and imprecise tasks that are designed to allow tradeoffs between the overall result quality and time and resource requirements [4-18].

IV. PERTS LANGUAGE AND ENVIRONMENT

We plan to build PERTS on networked workstations that support commercially available embedded software development tools, including a cross language system and cross debugging tools. The PERTS testbed will contain commercially available emulators so that it can provide a reasonably realistic execution environment for many embedded applications. The use of these tools will allow us to focus on the development of the tools and building blocks that are essential but are not currently available elsewhere, as well as the PERTS user interface and testbed. In addition, we plan to make the PERTS testbed configurable so that a variety of parallel and distributed hardware configurations can be simulated.

A key component of the PERTS user interface is the PERTS language used to describe the target task system to be analyzed and evaluated. The PERTS language is an extended C++ which combines the features of Flex and Real-Time Mentat, including the constraint and imprecision mechanisms in Flex and

the macro data-flow computation model of Real-Time Mentat. We have chosen to build PERTS based on C++ because we have already implemented many of the components of PERTS in C++ or an extended C++.

IV.1. The PERTS Program Model

A task system is represented in PERTS by its data-flow graph. Each node in the graph represents a task. At the most abstract level, a task is defined by its parameters: timing constraints, timing (and error) behaviors, and resource requirements. At the most concrete level, a task is defined by the code that implements it. A directed arc between two tasks *A* and *B* represents a dependency between the tasks. In particular, if there is an arc from *A* to *B*, then the execution of *B* cannot begin until the execution of *A* completes.

We note that at the abstract level, a data-flow graph is essentially the same as a precedence graph. Precedence graphs are typically used by the scheduler and resource access protocols to represent the task systems. Such a graph can be constructed automatically from models of application software components in a model base. For example, a data-flow graph description of a new task system can be obtained by linking subgraphs that characterize the individual building blocks of the system, and these subgraphs can be stored and maintained in the model base. PERTS will provide a visual programming interface via which a designer can provide the graphical description of the task system in this manner, generate new graphs or modify existing graphs. The abstract-level data-flow graph can be refined and made more and more concrete in later development phases when one wishes to have a task system simulated or emulated.

In an object-based prototyping environment, it would be easy to describe a prototype task system by its source code (and its specification in terms of the interfaces of objects in it and their interconnections). The PERTS language can serve this purpose in a C++ environment. At the most concrete level, the PERTS description of a task system is its source code in C++ plus the PERTS extensions and directives. We will return to discuss these extensions later in this section. These extensions are added to facilitate automatic detection of data flow and generation of data-flow graphs, to express the timing constraints of individual tasks, to provide scheduling directives for the execution environment, and to provide directives to the compiler for instrumenting the code. While the PERTS extensions can be used to annotate standard C++ programs, they in fact extend the C++ language into one that supports data-driven computation and provides the facilities for writing flexible real-time programs. The PERTS compiler will translate PERTS programs into C++ programs extended with library calls. We will use the PERTS language for the implementation and integration of tools.

The PERTS language is also based on the macro data-flow computation model first used in Mentat. Similar to the traditional large-grain, data-flow model, computation is data driven. A high-level view of each program is its data-flow graph, called a *program graph*. Tasks in this graph are also called *actors* in [11]. Each task is realized by an externally visible operation in the interface of some object. There is a directed arc from one task to another in the graph when there is a dependency between them. Tokens, containing data and control information, are sent along the arcs. When matching tokens on all its input arcs have arrived, a task is ready for execution. When a task executes, it consumes the input tokens. When its execution terminates, it generates one or more output tokens containing its result. The output tokens are forwarded along the arcs from it to the tasks that depend on the result.

The macro data-flow model differs from the traditional large-grain data-flow models in two ways. First, it allows for tasks which maintain state information between executions, called *persistent actors* in [11]. Persistent actors provide the ability lacking in the traditional models to model side effects and permit inter-program communication. This capability is necessary for applications such as data communication and database management. Second, while program graphs in traditional models are either entirely static or dynamic only to the extent of allowing a variable number of copies of some subgraphs, the structures of macro data-flow graphs can change dynamically as graph nodes are elaborated at run time into arbitrary subgraphs. Specifically, program graphs are represented using data structures called *futures* and *future lists*. Each task receives a future list with its input tokens; each future in the list names a dependent task that will receive a copy of the result produced by the task. The list sent to a dependent task describes a subgraph rooted at that task. Tasks may augment the future lists that are passed to them. When the execution of a task terminates, it sends a new future list, together with its result, to each of the dependent tasks named in its future list. In this way, the node representing a task in the program graph can be hierarchically elaborated and expanded. This feature allows us to describe dynamic programs whose data-flow-graph structures cannot be determined a priori at compile time in a way consistent with the descriptions of static programs and with the descriptions at the more abstract levels.

IV.2. The PERTS Programming Language

Each PERTS task may have its own timing constraints. The *timing blocks* and timing constraints extension is provided to express these constraints. The syntax of a timing block is as follows:

timing_constraint -> exception_action
begin statements **end;**

A timing constraint is a Boolean expression containing one or more of the following keywords: **start, deadline, duration,** and **period.** (They are, respectively, the time before which execution of the block must begin, the time at which its execution must end, the length of time interval during which the block may execute, the regular length of interval in which the block must execute once.) When the timing constraint expression evaluates to true, the exception action associated with it is executed.

In standard C++, member functions of an object are always available. **select/accept** statements are added to allow the programmer to specify those member functions that are candidates for execution based upon a broad range of criteria, including timing constraints of the computation. The semantics of the PERTS select statement is similar to the semantics of the select statement of ADA. A PERTS accept statement may list several alternative fct-declarators. The effect is that a call is made to one of the declared functions. Which function is called is decided at run time by the scheduler depending on the current timing constraints of the call. If one of the options can be completed in time while the others cannot, the one that can be is chosen. If more than one can be completed in time, the one listed first is chosen. In this way, the programmer can specify explicitly the different operations, with different timing requirements, that implement the same function.

In PERTS a timing constraint may be soft or hard and is considered hard unless it is declared to be otherwise by the keyword **soft.** The assumption here is: the underlying system ensures that all the hard timing constraints are met at all times and tries to maintain the soft timing constraints whenever it is

possible. In order for the system to be able to guarantee that hard timing constraints are met, graph elaboration must be restricted for tasks with this type of constraint. Moreover every task (and subgraph) with a hard timing constraint must have bounded execution time and resource requirements that can be determined from static analysis at compile time. This makes it possible for the schedulability of these tasks and the strategy used to schedule them to be determined at compile time. A task (or subgraph) with a soft timing constraint, on the other hand, can be elaborated dynamically at run time into different subgraphs. When the imprecise computation technique is used, each task with a hard timing constraint is expanded into two parts: its mandatory part and its optional part; they have hard and soft timing constraints, respectively.

Some of the other PERTS extensions are the keyword **Perts** in class descriptions; **create()/destroy()** statements; and return to future (**rtf()**) statement. The function **rtf()** is the PERTS analog to the **return()** of C++. Its purpose is to allow each PERTS member function (task) to return a value to its successor nodes in the program graph. A legal exception of a timing block is **rtf()**. This makes it possible to return an immediate result produced so far in the course of the computation if the execution of the timing block must be prematurely terminated.

To provide the programmer with a way to control the granularity and, hence, the degree of parallelism, PERTS allows both standard C++ classes and **Perts** classes to be defined. By default, a standard C++ class definition defines a standard C++ class. A **Perts** class has the keyword **Perts** in the class definition. It defines a set of tasks, one for each external operation, that can potentially be executed in parallel with other tasks in order to improve performance. Classes of objects whose external operations are not sufficiently complex and cannot be efficiently executed in parallel with other such operations due to high communication overhead should be defined as standard C++ classes. The two new reserved member functions for all Perts class objects, **create()** and **destroy()**, are provided so that generic instances (that is, instances of Perts classes represented by their names rather than their physical classes) can be defined [11]. **create()** allows the programmer to provide optional location hints (e.g., high_communication_ratio, co_locate, etc.) that inform the underlying system where, ideally, the new instance is to be instantiated, e.g., on a different processor or on the same processor as some other Perts objects. The location hints allow the programmer to influence the underlying system's decision on where the new object is actually instantiated. Both **Perts** and the **create()/select()** statements can be used as annotations in standard C++ programs to provide help or specify restrictions on how the task system is to be scheduled.

IV.3. The PERTS Virtual Machine

We plan to build the PERTS execution environment based on the Real-Time Mentat virtual machine architecture. To support parallel execution of programs on a multiprocessor or distributed system, a virtual machine runs on each processor. The virtual machine is based on the existing Mentat virtual machine, which has been implemented on the Encore Multimax as well as hypercubes and networked Sun workstations [8]. Conceptually, the structure of the Mentat virtual machine is similar to that of the traditional data-flow machine.

The Real-Time Mentat virtual machine differs from existing ones in the ways that tokens are sent, the ready-to-run queue is ordered, and the matching and computation units operate. In Real-Time

Mentat, the update unit may preferentially forward and update the results from computations according to the underlying scheduling policy. Similarly, the ready-to-run queue is ordered according to the scheduling policy that is being evaluated. The matching unit keeps track of the timing constraints of tasks who have not yet received all of their input tokens and ensures the appropriate exception handling if any of these constraints is violated. This architecture will provide us with a run-time environment suitable for simulating many different system configurations.

Again, within the PERTS execution environment, it will be possible for the designer to select and to assemble building blocks for real-time systems, down load them to an emulated or simulated platform, and test them in a reasonably realistic manner. To provide a flexible environment in which representative embedded environments and workloads can be simulated, a workload simulator will be needed. Recently, real-time workload benchmarks that characterize several typical embedded systems and applications (for example, avionics and navigation, signal processing, and robotics) have been proposed (e.g, [34-36]). As a starting point, our simulator will generate these synthetic workloads.

V. RELATED WORK

PERTS is similar to Scheduler 1-2-3 [37] and many other real-time systems design tools [37-39] in their intended use. These systems all intend to reduce the complexity in real-time system development. The advanced algorithms and tools available in PERTS distinguish it from the other systems. For example, the tool set in Scheduler 1-2-3 consists of a schedulability analyzer and a real-time monitor/debugger for the ARTS Kernel, an operating system built at Carniege-Mellon University. The Scheduler 1-2-3 schedulability analyzer primarily deals with periodic tasks mixed with randomly arriving aperiodic tasks, the kind of workload typically encountered in traditional real-time applications. However, only priority-driven scheduling disciplines are supported in Scheduler 1-2-3. Several systems similar to Scheduler 1-2-3 are also available. They are used to support the design and construction of specific applications on a specific hardware platform. PERTS, on the other hand, provides a much more versatile and powerful schedulability analysis system. The PERTS testbed can be configured to simulate a number of operating systems and hardware configurations. It will be an ideal environment for doing the kind of experiments that will need to be done during technology insertion.

PERTS differs from most existing and experimental real-time system prototyping and development systems, and complements them, both in their capabilities and intended use. Many such systems typically provide an integrated environment with a full range of software prototyping and development tools for requirement tracing, program construction, software reuse, software and tool integration, etc. For example, the experimental system CAPS [39] supports prototyping of complex real-time software systems. PERTS is similar to CAPS in certain ways; for instance, both provide tools for analysis and debugging real-time software. The PERTS language serves similar purposes as the Prototype System Description Language (PSDL) in CAPS. However, PSDL is a special-purpose language. The PERTS language, on the other hand, is an extended C++ language with a small number of simple extensions. CAPS is a stand-alone prototyping environment. PERTS is not designed to be a substitute for CAPS or other computer-aided software prototyping systems. Rather, PERTS will focus on providing powerful real-time system design and evaluation tools that are not available in these systems.

VI. SUMMARY

Future real-time systems must be easy to modify and to maintain. They must guarantee that real-time tasks can be scheduled to meet their timing constraints. However, current software technology often treats scheduling as an after-thought. Often it is too late for the systems to be correctly implemented. PERTS supports an integrated approach to prototyping in which the user considers, and gets feedback on, the schedulability of the target system early in the design process. As a result, the system implemented can have a verifiable performance yet is easy to modify and to extend.

The goal of this project is to make the recent and future theoretical advances in real-time system design and engineering readily usable to practitioners. PERTS will provide an environment for the evaluation of new design approaches, for experimentation with alternative system building blocks, and for analysis and performance profiling of prototype real-time systems. PERTS is not intended for any particular application domain and is not designed to serve as a stand-alone rapid prototyping environment. Rather, PERTS as a whole, as well as the individual tools and the building blocks provided by it, will be made easily integratable into most object-based software engineering environments for building real-time systems. In this way, we believe, PERTS can make its best contribution to the real-time systems technology.

REFERENCES

[1] Liu, C. L. and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the Association for Computing Machinery, Vol. 20, No. 1*, pp. 46-61, January 1973.

[2] Dhall, S. K. and C. L. Liu, "On a real-time scheduling problem," *Operations Research*, Vol. 26, No. 1, pp. 127-140, February 1978.

[3] Shih, W. K., J. W. S. Liu, and C. L. Liu, "Modified rate-monotone algorithm for scheduling periodic jobs," submitted.

[4] Liu, J. W. S., K. J. Lin, C. L. Liu, "Concord prototype system and real-time scheduling," *Proceedings of 4th Workshop on Real-Time Operating Systems*, Cambridge, MA, pp. 27-30, July 1987.

[5] Lin, K. J., S. Natarajan, J. W. S. Liu, and T. Krauskopf, "Concord: A system of imprecise computations," *Proceedings of 11th Computer Software & Application Conference* (COMPSAC), Tokyo, Japan, pp. 75-81, October 1987.

[6] Lin, K. J., S. Natarajan, and J. W. S. Liu, "Imprecise results: utilizing partial computations in real-time systems," *Proceedings of Eighth Real-Time Systems Symposium*, pp. 210-217, San Jose, CA, December 1987.

[7] Liu, J. W. S. and K. J. Lin, "On means to provide flexibility in scheduling," *Proceedings of 2nd International Workshop on Real-Time Ada Issues*, Devon, England, pp. 96-105, June 1988.

[8] Natarajan, S. and K. J. Lin, "FLEX: toward flexible real-time programs," *Proceedings of IEEE International Conference on Computer Languages '88*, Miami Beach, FL, pp. 272-279, October 1988.

[9] Natarajan, S. and K. J. Lin, "Expressing and maintaining timing constraints in FLEX," *Proceedings of 9th Real-Time Systems Symposium*, Huntsville, Alabama, pp. 96-105, December 1988.

[10] Grimshaw, A., J. W. S. Liu and A. Silberman, "Real-Time Mentat, a data-driven, object-oriented system," *Proceedings of IEEE Globecom*, pp. 141-147, Dallas, Texas, November 1989.

[11] Grimshaw, A., J. W. S. Liu and A. Silberman, "Real-Time Mentat programming language and architecture," *IEEE Real-Time Systems Newsletter*, Special Issue on 7th IEEE Workshop on Real-Time Operating Systems and Software, pp. 82-87. Charlottesville, Virginia, Spring 1990.

[12] Liu, J. W. S., K. J. Lin, C.L. Liu, and C. W. Gear, "Research on imprecise computation," *Proceedings of Workshop on Operating Systems for Mission Critical Computing*, College Park, MD, September 1989.

[13] Kenny, K. B. and K. J. Lin, "Implementing real-time systems using performance polymorphism," *Proceedings of Compsac 90*, Chicago, IL, pp. 693-698, October 1990.

[14] Natarajan, S. and K. J. Lin, "FLEX: Toward flexible real-time programs," *Computer Language Journal*. Pergamon Press, New York, 1990.

[15] Kenny, K. B. and K. J. Lin, "Structuring large real-time systems with performance polymorphism," *Proceedings of 11th IEEE Real-Time Systems Symp.*, Orlando, FL, pp. 238-246, December 1990.

[16] Gregory, G. J. and K. J. Lin, "Building real-time imprecise computations in Ada," *Proceedings of TriAda '90*, Baltimore, MD, ACM, December 1990.

[17] Kenny, K. B. and K. J. Lin, "Building flexible real-time systems using the FLEX language," *IEEE Computer*, Special Issue on Real-Time Systems, May 1991.

[18] Kenny, K. B. and K. J. Lin, "A measurement-based performance analyzer for real-time programs," *Proceedings of Int. Phoenix Conf. on Computers and Communications*, Scottsdale, AZ, March 1991.

[19] Liu, J. W. S., S. Natarajan, and K. J. Lin, "Scheduling real-time, periodic jobs using imprecise results," *Proceedings of Eighth Real-Time Systems Symposium*, pp. 252-260, San Jose, CA, December 1987.

[20] Chung, J. Y. and J. W. S. Liu, "Algorithm for scheduling periodic jobs to minimize average error," *Proceedings of the 9th IEEE Real-Time Systems Symposium* Huntsville, Alabama, December 1988.

[21] Chung, J. Y. and J. W. S. Liu, "Performance of algorithms for scheduling periodic jobs to avoid timing faults," *Proceedings of the 22nd Hawaii International Conference on System Sciences*, Kailua-Kona, Hawaii, January 1989.

[22] Shih, W. K., J. Y. Chung, J. W. S. Liu, and D. W. Gillies, "Scheduling tasks with ready times and deadlines to minimize average error," *ACM Operating Systems Review*, pp. 14-28, July 1989.

[23] Chung, J. Y., W. K. Shih, J. W. S. Liu and D. W. Gillies, "Scheduling imprecise computations to minimize total error," *Microprocessing and Microprogramming*, 27, pp. 767-774, 1989.

[24] Shih, W. K., J. W. S. Liu and J. Y. Chung, "Fast algorithms for scheduling tasks with ready times and deadlines to minimize total error," *Proceedings of the 10th IEEE Real-Time Systems Symposium*, pp. 12-19, Los Angeles, California, December 1989.

[25] Han, C. and K. J. Lin, "Scheduling parallelizable jobs on multiprocessors," *Proceedings of 10th Real-Time Systems Symposium*, Santa Monica, CA, Dec. 1989.

[26] Yu, A. and K. J. Lin, "Scheduling parallelizable imprecise computations on multiprocessors," *Proceedings of 5th IEEE Int. Parallel Processing Symposium*, Anaheim, CA, April 1991.

[27] Liu, J. W. S., K. J. Lin, W. K. Shih, A. C. Yu, J. Y. Chung, and W. Zhao, "Algorithms for scheduling imprecise computations," *IEEE Computer*, Special Issue on Real-Time Systems, May 1991.

[28] Shih, W. K., J. W. S. Liu and J. Y. Chung, "Algorithms for scheduling tasks to minimize total error," to appear in *SIAM Journal of Computing*, July 1991.

[29] Smith, K. P. and J. W. S. Liu, "Monotonically improving approximate answers to relational algebra queries," *Proceedings of IEEE 1989 Compsac*, pp. 234-241, September 1989.

[30] Vrbsky, S. and J. W. S. Liu, "An object-oriented query processor that returns monotonically improving answers," *Proceedings of 7th IEEE Conf. on Data Engineering*, Kobe, Japan, April 1991.

[31] Gillies, D. and J. W. S. Liu, "The Complexity of AND/OR Scheduling," *Proceedings of the 2nd IEEE Conference on Parallel and Distributed Processing*, pp. 394-401, Dallas, Texas, December 1990.

[32] Han, C. and K. J. Lin, "Scheduling jobs with temporal consistency constraints," *Proceedings of 6th Workshop on Real-Time Software and Operating Systems*, Pittsburgh, PA, May 1989.

[33] Bettati, R. and J. W. S. Liu, "End-to-end scheduling to meeting deadlines," *Proceedings of the 2nd IEEE Conference on Parallel and Distributed Processing*, pp. 62-67, Dallas, Texas, Dec. 1990.

[34] Tokuda, H. "Distributed Hartstone," *Proceedings of the 10th International Conference on Distributed Computing Systems*, Paris, France, June 1990.

[35] Weiderman, N., "Hartstone: A Series of Hard Real-Time Benchmarks," Technical Report No. CMU/SEI-89-TR-23, Software Engineering Institute, Carnegie Mellon University, June 1989.

[36] Molini, J., S. Maimon and P. Watson, "Real-Time System Scenarios", *Proceedings of RTSS*, December 1990.

[37] Tokuda, H. and C. W. Mercer, "A Real-Time Tool Set for the ARTS Kernel," *Proceedings of the 9th IEEE Real-Time Systems Symposium*, December 1988.

[38] Kiskis, D. L. and K. G. Shin, "A Synthetic Workload for Real-Time Systems," *Proceedings of the 7th IEEE Workshop on Real-Time Operating Systems and Software*, Charlottesville, Virginia, May 1990.

[39] Luqi, "Software Evolution Through Rapid Prototyping," *IEEE Computer*, May 1989.

54-61
93856
245135
18 p.

Issues in the Imprecise Computation Approach to Fault Tolerance

Jane W. S. Liu, Kwei-Jay Lin and C. L. Liu
Department of Computer Science
University of Illinois

ABSTRACT

The imprecise computation technique can be used in a natural way to enhance fault tolerance. By providing a usable, approximate result whenever a failure or overload prevents the system from producing the desired, precise result, we can increase the availability of data and services, reduce the need for error-recovery operations, and minimize the costs in replication. This paper describes the domain-specific fault tolerance mechanisms that are needed to support the provision and correct usage of imprecise results for several representative application domains. The elements of an application-domain-independent architecture that can effectively integrate these domain-specific mechanisms are also described.

I. INTRODUCTION

The *imprecise computation technique* [1-15] has been proposed in 1987 as a way to provide flexibility in scheduling time-critical tasks. A system based on this technique makes an acceptable approximate result available whenever an overload condition prevents an exact result of the desired quality from being produced in time. In this way, the difficulty in meeting all deadlines at all times in dynamic systems is minimized. Examples of applications where one may prefer timely, approximate results, to late, exact results are image processing and tracking. It is often better to have, in time, frames of fuzzy images or estimates of target locations than perfect images or accurate location data too late. Since the advent of the imprecise computation technique, significant progress has been made in imprecise-computation scheduling. Several workload models have been developed to characterize imprecise computations. Many scheduling algorithms have been developed, including algorithms for scheduling imprecise jobs on uniprocessors, jobs with multiple versions, and parallelizable and multitasking computations on multiprocessor systems [16-33]. Several approaches to implementing imprecise computations in Flex and Real-Time Mentat [6-9,11-13,15], as well as in Ada [5,14], have been proposed. The former two are experimental, extended C++ programming languages designed for writing flexible programs based on the imprecise computation model.

We note that the imprecise computation technique can also be used in a natural way to enhance fault tolerance and graceful degradation. Providing usable, imprecise results whenever a failure or overload prevents the system from producing the desired, precise results is a way to increase the availability of data and services, to reduce the need for error-recovery operations, to simplify the recovery operations that must be done, and to reduce the costs in replication. Here by faults or failures, we mean hardware faults and failures, as well as transient software faults that disappear after the system software is reset to a fault-free state. Permanent software faults are not considered.

The imprecise computation approach to fault tolerance is application-domain dependent. Imprecise results can be used correctly as approximations of precise ones in an application domain only when the semantics of approximation is well defined and the effect of inaccuracy can be rigorously bounded. Most

In *Proceedings of 1991 System Design Synthesis Technology Workshop*, Silver Spring, MD, September 1991.

Conference, Portland, Oregon, pp. 562-569, April 28-May 1, 1991.

NASA
SB-62
93861
245126
8P.

Scalable Performance Environments for Parallel Systems

Daniel A. Reed*, Robert D. Olson, Ruth A. Aydt, Tara M. Madhyastha,
Thomas Birkett, David W. Jensen, Bobby A. A. Nazief, and Brian K. Totty

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

Abstract

As parallel systems expand in size and complexity, the absence of performance tools for these parallel systems exacerbates the already difficult problems of application program and system software performance tuning. Moreover, given the pace of technological change, we can no longer afford to develop *ad hoc*, one-of-a-kind performance instrumentation software; we need scalable, portable performance analysis tools. We describe an environment prototype based on the lessons learned from two previous generations of performance data analysis software. Our environment prototype contains a set of performance data transformation modules that can be interconnected in user-specified ways. It is the responsibility of the environment infrastructure to hide details of module interconnection and data sharing. The environment is written in C++ with the graphical displays based on X windows and the Motif toolkit. It allows users to interconnect and configure modules graphically to form an acyclic, directed data analysis graph. Performance trace data are represented in a self-documenting stream format that includes internal definitions of data types, sizes, and names. The environment prototype supports the use of head-mounted displays and sonic data presentation in addition to the traditional use of visual techniques.

1 Introduction

As the class of high-performance computer systems extends beyond traditional vector supercomputers, there is growing concern about performance variance. For sequential systems, increasing clock rates, MIPS, and MFLOPS are raising the performance of almost all applications. In contrast, the expected performance for a suite of parallel applications can vary widely, depending on the match of parallel architecture, system software, and application algorithms [11]. With the projected emergence of massively parallel MIMD systems (e.g., the Intel Touchstone project goals include a 2048 node system based on i860 microprocessor technology [7]), understanding and correcting application and system performance bottlenecks become crucial.

Although it is now architecturally feasible to construct scalable parallel systems that contain thousands of processors, there is no discipline or set of standard software libraries for evaluating the performance of these systems or their application programs. Moreover, almost all extant performance environments are one-of-a-kind implementations for specific, modest scale parallel systems; they do not address the more difficult issues of scalable performance measurement and analysis. As parallel systems expand in size and complexity, the absence of performance tools for these scalable parallel systems exacerbates the already difficult problems of application program and system software performance tuning. Moreover, given the pace of technological change, we can no longer afford to develop *ad hoc*, one-of-a-kind performance instrumentation software; we need scalable, portable performance analysis tools.

Clearly, performance instrumentation is a means, not an end (i.e., the intent of performance instrumentation is user insight and program performance optimization). Repeated experience has shown that if performance analysis tools are difficult to use, software developers will eschew these tools in favor of simpler, inferior techniques, even if it means the concomitant abandonment of a sophisticated performance data analysis environment. In [8], we argued that integration and flexibility are the twin keys to an effective performance analysis environment. If the performance instrumentation environment does not permit diverse approaches to performance data reduction and analysis, its functional lifetime will be limited.

In [13, 9, 10], we reported the completion of performance data visualization prototypes that supported both dynamic graphic displays and event timelines. A careful analysis of this "proof of concept" implementation has provided many valuable lessons that have driven our design of a new data visualization and sonification system.

*Supported in part by the National Science Foundation under grants NSF CCR87-06653 and NSF CDA87-22836, by the National Aeronautics and Space Administration under NASA Contract Number NAG-1-613, and by grants from Apple Computer and the Digital Equipment Corporation External Research Program.

updates gives a possible method for generating the result models from the requirements and constraints of the databases.

V. RELATION TO TRADITIONAL APPROACHES

We can view the imprecise computation technique as a way to increase the availability of critical data and services. Traditionally, high availability of data and services is maintained in dependable systems by replication (for examples [50-54]). By storing duplicate copies of data objects, they are more likely to remain available when processor or network failures make some copies inaccessible. Similarly, computations can be made more resilient by replicated execution of the same programs on multiple computers. In contrast, the imprecise computation technique increases availability of data objects and services by making sure that imprecise results, approximate versions of data objects and degraded levels of services, of sufficiently good quality are available whenever it is feasible to do so. The obvious advantage of the imprecise computation approach is its low cost compared with traditional replication approaches. However, it is a viable alternative to the traditional approaches only for applications that can make use of imprecise results and degraded services in place of the desired ones. For these applications, the imprecise computations can be used together with replication to reduce the overhead of replication. As discussed earlier, monotone imprecise computations make partial replication possible; only the mandatory portion of each task is replicated.

The imprecise computation technique offers a new dimension to the traditional error recovery methods (for examples, [34-37]). In an imprecise system, when a server terminates prematurely due to a crash, an imprecise result will be returned. Error recovery is only necessary when the imprecise result is not acceptable. Reducing the need for error recovery is a distinct advantage of the imprecise computation approach. This advantage is especially important in distributed systems, where the cost of error recovery is often high, and in hard real-time systems, where there is often no time for error recovery.

For many applications, a series of imprecise results produced by a computation can allow the extrapolation of a good estimate of the final result of the computation. It may be even possible to predict the final result based on the history giving the past imprecise results. Such forward recovery methods offer a distinct advantage over other methods.

We can view the imprecise computation approach as a semantical approach to checkpointing. An imprecise result produced by a task after the completion of its mandatory portion is in fact a checkpoint that happens to be usable as it is by the application. Together with the accuracy indicators, they may provide the completion state information or only partial state information. Some of the issues in semantical checkpointing are also similar to the ones that have arisen in semantic concurrency control: how to allow programmer specified correctness criteria, integrate data and operation semantics, etc.

VI. SUMMARY

We have discussed the key problems in the imprecise computation approach to enhance fault tolerance and graceful degradation. We have also described our ongoing activities in the development of an imprecise system architecture. This architecture supports imprecise service establishment and result saving protocols for the provision and use of imprecise results, imprecision management policies to ensure correct usage of imprecise results, and error recovery making use of imprecise results. To make

the imprecise computation technique readily applicable, domain-specific mechanisms will be needed to fit into this architectural framework. Elements of domain-specific mechanisms include meaningful and usable semantics for interpreting imprecise results as approximations of precise results, quality metrics for comparing the approximations, and monotone computational algorithms to support these approximation semantic and quality metrics. We are currently developing these elements for several key application domains, including database queries and updates, numerical computations, signal processing, and control.

REFERENCES

[1] Liu, J. W. S., K. J. Lin and C. L. Liu, "A position paper for the IEEE 1987 Workshop on Real-Time Operating Systems," Cambridge, Mass, May, 1987.

[2] Lin, K. J., S. Natarajan, J. W. S. Liu and T. Krauskopf, "Concord: a system of imprecise computations," *Proceedings of the 1987 IEEE Compsac*, pp. 75-81, Tokyo, Japan, October 7-9, 1987.

[3] Lin, K. J., S. Natarajan, J. W. S. Liu, "Imprecise results: utilizing partial computations in real-time systems," *Proceedings of the IEEE 8th Real-Time Systems Symposium*, San Jose, California, December 1987.

[4] Natarajan, S. and K. J. Lin, "Refinement and enhancement: primitives for monotonic computations," *Proceedings of 5th Workshop on Real-Time Software and Operating Systems*, Washington, DC, pp. 27-31, May 1988.

[5] Liu, J. W. S. and K. J. Lin, "On means to provide flexibility in scheduling," *Proceedings of 2nd International Workshop on Real-Time Ada Issues*, Devon, England, pp. 96-105, June 1988.

[6] Natarajan, S. and K. J. Lin, "FLEX: toward flexible real-time programs," *Proceedings of IEEE International Conference on Computer Languages '88*, Miami Beach, FL, pp. 272-279, October 1988.

[7] Natarajan, S. and K. J. Lin, "Expressing and maintaining timing constraints in FLEX," *Proceedings of 9th Real-Time Systems Symposium*, Huntsville, Alabama, pp. 96-105, December 1988.

[8] Grimshaw, A., J. W. S. Liu and A. Silberman, "Real-Time Mentat, a data-driven, object-oriented system," *Proceedings of IEEE Globecom*, pp. 141-147, Dallas, Texas, November 1989.

[9] Grimshaw, A., J. W. S. Liu and A. Silberman, "Real-Time Mentat programming language and architecture," *IEEE Real-Time Systems Newsletter*, Special Issue on 7th IEEE Workshop on Real-Time Operating Systems and Software, pp. 82-87. Charlottesville, Virginia, Spring 1990.

[10] Liu, J. W. S., K. J. Lin, C.L. Liu, and C. W. Gear, "Research on imprecise computation," *Proceedings of Workshop on Operating Systems for Mission Critical Computing*, College Park, MD, September 1989.

[11] Kenny, K. B. and K. J. Lin, "Implementing real-time systems using performance polymorphism," *Proceedings of Compsac 90*, Chicago, IL, pp. 693-698, October 1990.

[12] Natarajan, S. and K. J. Lin, "FLEX: Toward flexible real-time programs," *Computer Language Journal*. Pergamon Press, New York, 1990.

[13] Kenny, K. B. and K. J. Lin, "Structuring large real-time systems with performance polymorphism," *Proceedings of 11th IEEE Real-Time Systems Symp.*, Orlando, FL, pp. 238-246, December 1990.

[14] Gregory, G. J. and K. J. Lin, "Building real-time imprecise computations in Ada," *Proceedings of TriAda '90*, Baltimore, MD, ACM, December 1990.

[15] Kenny, K. B. and K. J. Lin, "Building flexible real-time systems using the FLEX language," *IEEE Computer, May 1991.*

[16] Liu, J. W. S., S. Natarajan, and K. J. Lin, "Scheduling real-time, periodic jobs using imprecise results," *Proceedings of Eighth Real-Time Systems Symposium*, pp. 252-260, San Jose, CA, December 1987.

[17] Chung, J. Y. and J. W. S. Liu, "Algorithm for scheduling periodic jobs to minimize average error," *Proceedings of the 9th IEEE Real-Time Systems Symposium* Huntsville, Alabama, December 1988.

[18] Chung, J. Y. and J. W. S. Liu, "Performance of algorithms for scheduling periodic jobs to avoid timing faults," *Proceedings of the 22nd Hawaii International Conference on System Sciences*, Kailua-Kona, Hawaii, January 1989.

[19] Chung, J. Y., "Scheduling periodic jobs making use of imprecise results," Ph.D. Thesis, Department of Computer Science, University of Illinois, May 1989.

[20] Shih, W. K., J. Y. Chung, J. W. S. Liu, and D. W. Gillies, "Scheduling tasks with ready times and deadlines to minimize average error," *ACM Operating Systems Review*, pp. 14-28, July 1989.

[21] Chung, J. Y., W. K. Shih, J. W. S. Liu and D. W. Gillies, "Scheduling imprecise computations to minimize total error," *Microprocessing and Microprogramming*, 27, pp. 767-774, 1989.

[22] Shih, W. K., J. W. S. Liu and J. Y. Chung, "Fast algorithms for scheduling tasks with ready times and deadlines to minimize total error," *Proceedings of the 10th IEEE Real-Time Systems Symposium*, pp. 12-19, Los Angeles, California, December 1989.

[23] Leung, J. Y-T., T. W. Tam, C. S. Wong, and G. H. Wong, "Minimizing mean flow time with error constraints," *Proceedings of the 10th IEEE Real-Time Systems Symposium*, December 1989.

[24] Han, C. and K. J. Lin, "Scheduling imprecise computations on parallel real-time systems," with C. Han, *Proceedings of 4th Annual Symposium on Parallel Processing*, Fullerton, CA, April 1990.

[25] Chung, J. Y., J. W. S. Liu, and K. J. Lin, "Scheduling periodic jobs that allow imprecise results," *IEEE Transactions on Computer*, Vol. 39, No. 9, pp. 1156-1174, September 1990.

[26] Leung, J. Y-T. and C. S. Wong, "Minimizing the number of late tasks with error constraints," *Proceedings of the 11th Real-Time Systems Symposium*, Orlando, Florida, December 1990.

[27] Yu, A. and K. J. Lin, "Scheduling parallelizable imprecise computations on multiprocessors," *Proceedings of 5th IEEE Int. Parallel Processing Symposium*, Anaheim, CA, April 1991.

[28] Liu, J. W. S., K. J. Lin, W. K. Shih, A. C. Yu, J. Y. Chung, and W. Zhao, "Algorithms for scheduling imprecise computations," *IEEE Computer, May 1991.*

[29] Shih, W. K., J. W. S. Liu and J. Y. Chung, "Algorithms for scheduling tasks to minimize total error," to appear in *SIAM Journal of Computing*, July 1991.

[30] Tu, P. and K. J. Lin, "Scheduling performance polymorphic computations in real-time systems," *submitted for publication.*

[31] Yu, A. and K. J. Lin, "A scheduling algorithm for fault tolerant imprecise computations on multiprocessors," *Proc. 5th IEEE Int. Parallel Processing Symposium*, Anaheim, CA, April 1991.

[32] Zhao, W. and J. W. S. Liu, "An analytical model for multi-server imprecise systems", submitted

[33] Shih, W. K. and J. W. S. Liu, "Minimization of the maximum error of imprecise computations." submitted

[34] Campbell, R.H., and Randell, B., "Error recovery in asynchronous systems," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 811-826, Aug. 1986.

[35] Koo, B. and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Transactions on Software Engineering*, January 1987.

[36] Leu, P. and B. Bhargava, "Concurrent robust checkpointing and recovery in distributed systems," *Proceedings of the 4th IEEE International Conference on Data Engineering*, 1988.

[37] Tamir, Y. and C. H. Sequin, "Error recovery in multicomputers using global checkpoints," *Proceedings of the 13th International Conference on Parallel Processing, 1984.*

[38] Wallace, G. K., "Overview of the JPEG (ISO/CCITT) still image compression standard," Visual Communication and Image Processing, '89, SPIE, Philadelphia, November 1989.

[39] Motro, Amihai, "VAGUE: a user interface to relational databases that permits vague queries," *ACM Transactions on Office Information Systems*, Vol. 6, No. 3, pp. 187-214, July 1988.

[40] Liu, K. and R. Sunderraman, "On representing indefinite and maybe information in relational databases," *Proceedings of the 4th International Conference on Data Engineering*, pp. 250-257, February 1988.

[41] Ichikawa, T. and M. Hirakawa, "ARES: a relational database with the capability of performing flexible interpretation of queries," *IEEE Transactions on Software Engineering*, Vol.SE-12, No.5, pp. 624-634, May 1986.

[42] Imielinski, T and W. Lipski, "Incomplete information and dependencies in relational databases," *Proceedings of the ACM SIGMOD Conference on Data Management*, pp. 178-184, May 1983.

[43] Abiteboul, S. and G. Grahne, "Update semantics for incomplete databases," *Proceedings of the 11th International Conference on VLDB*, Stockholm, pp. 1-12, 1985.

[44] Reiter, R., "A sound and sometimes complete query evaluation algorithm for relational database with null values," *JACM*, Vol. 33, No. 2, pp. 349-370, April 1986.

[45] Winslett, M., "A model-based approach to updating databases with incomplete information," *ACM Transactions on Database Systems*, Vol.13, No.2., pp. 167-196, June 1988.

[46] Ozsoyoglu, G., Z. M. Ozsoyoglu, and W. Hou, "Research in time- and error-constrained database query processing," *Proceedings of the 7th IEEE Workshop on Real-Time Operating Systems and Software*, Charlottesville, VA, May 1990.

[47] Smith, K. P. and J. W. S. Liu, "Monotonically improving approximate answers to relational algebra queries," *Proceedings of IEEE 1989 Compsac*, pp. 234-241, September 1989.

[48] Vrbsky, S. and J. W. S. Liu, "An object-oriented query processor that returns monotonically improving answers," *Proceedings of 7th IEEE Conf. on Data Engineering*, Kobe, Japan, April 1991.

[49] Buneman, P., S. Davidson, and A. Watters, "A semantics for complex objects and approximate queries," *Proceedings of the Seventh Symposium on the Principles of database Systems*, pp. 305-314, March 1988.

[50] Ahamad, M., P. Dasgupta, R. LeBlanc and C. T. Wilkes, "Fault tolerant computing in object-based distributed operating systems," *Proceedings of the 6th IEEE Symposium on Reliable Distributed Systems*,, 1987.

[51] Cooper, E. C., "Replicated distributed programs," *Proceedings of the 10th Symposium on Operating Systems Principles*, ACM, 1985.

[52] Ng, T. P. and S. B. Shi, "Replicated transactions," *Proceedings of IEEE 9th International Conferences on Distributed Computing Systems*, 1989.

[53] Popek, G. J. and B. J. Walker, *The LOCUS Distributed System Architecture*, the MIT Press, 1985.

[54] Obradovic, M. and P. Berman, "Voting as the optimal static pessimistic scheme for managing replicated data," *Proceedings of the 9th IEEE Symposium on Reliable Distributed Systems*, October 1990.

NASA
56-61
73866

245137
15P.

Experiences Building an Object-Oriented System in C++*

Peter W. Madany and Roy H. Campbell and Panagiotis Kougiouris
University of Illinois at Urbana-Champaign
Department of Computer Science
1304 W. Springfield Avenue, Urbana, IL 61801

Abstract

This paper describes tools that we built to support the construction of an object-oriented operating system in C++. The tools provide the automatic deletion of unwanted objects, first-class classes, dynamically loadable classes, and class-oriented debugging. As a consequence of our experience building *Choices*, we advocate these features as useful, simplifying and unifying many aspects of systems programming.

1 Introduction

Choices[2] is an object-oriented operating system written in an object-oriented language (C++[13]). It supports an object-oriented application interface based on objects, inheritance, and polymorphism. User and system objects can be created and deleted dynamically. Objects may be persistent and have a global lifetime. The system supports the debugging of objects that are instances of a class. In this paper we describe our experiences programming the system and the tools we built to take advantage of C++ in the construction of a large object-oriented system. These tools include:

- the automatic deletion of unreferenced objects from various environments,
- run-time exposure of classes and the class hierarchy,
- facilities to load dynamically new subclasses and code into the kernel or a user program,
- debugging facilities for objects and classes that are vendor-, machine-, and environment-independent.

Our research efforts are directed towards developing an object-oriented operating system and not towards language design. We seek to use the object-oriented facilities of a language like C++ to achieve our goals of building an object-oriented operating system. From our perspective, an object-oriented systems programming language should include a minimal set of efficient encapsulation, inheritance, and polymorphism mechanisms that can be used to build the other concepts required in developing a system. Although C++ itself is not a minimal language, in our experience we have found it efficient[11]. Although certain object-oriented facilities are not present in the language, we have been able to construct these facilities from more primitive mechanisms. We believe our experiences may be useful to future object-oriented system language designers and builders of other object-oriented systems.

Section 2 introduces the *Choices* operating system and provides the motivation for developing garbage-collection, first-class classes, run-time loadable code, and portable debugging facilities. The next four sections describe our implementation of each of these features. Section 7 summarizes our experiences.

*This work was supported in part by NSF grant CISE-1-5-30035 and by NASA grants NSG1471 and NAG 1-163.

systems encountered in practice must support applications in many domains. Therefore, there must be an architectural framework for the effective integration of the domain-specific, fault tolerance mechanisms built on the imprecise computation technique.

This paper discusses the problems that must be solved to make the imprecise computation technique a feasible and effective means to fault tolerance. It also describes our ongoing efforts and future work in this direction. The remainder of the paper is organized as follows. Section II provides the background and motivation, as well as introducing the terms used in the paper. Section III describes the key components of a system architecture designed to provide and use imprecise computations to enhance availability and fault tolerance. Section IV discusses the domain-specific issues. Section V discusses the relationship between the imprecise computation approach and the traditional approaches to fault tolerance. Section VI is a summary.

II. BACKGROUND AND MOTIVATION

We call a system that supports imprecise computations an *imprecise system.* In an imprecise system, each critical task is structured in such a way that it can be logically decomposed into two subtasks: a mandatory subtask and an optional subtask. By a *task*, we mean a granule of computation (or communication) that is treated as an individual unit of work; hereafter, we will use the terms task and computation (or communication) interchangeably. The mandatory subtask is the portion of the computation (or communication) that must be completed in order for the task to produce a result of an acceptable quality. The optional subtask is the portion that refines the result. The optional subtask can be left unfinished and terminated, if necessary, with an accompanied reduction in the result quality.

The result produced by a task when it completes is the desired, *precise*, one. If the task is terminated before it completes, the approximate result produced at the time of its termination is *acceptable* and usable as long as its mandatory subtask is completed. Such a result is said to be *imprecise*; it is an *approximation* of the precise result. The *accuracy* of a precise result is equal to one, the maximum possible value. The accuracy of an imprecise result is less than one; it is typically a function of the execution time of the completed portion.

There are three methods for providing imprecise results: the sieve, multiple-version, and milestone methods [3,4,11-13]. The method ideally suited for the fault tolerance purpose is the milestone method. The *milestone method* requires the computations to be monotone. A task, as well as the underlying computational algorithm, is said to be *monotone* if the quality of the intermediate result produced by it is non-decreasing as it executes longer. An imprecise result of a monotone task can be made available by recording the intermediate results produced by the task at appropriate instances of its execution. The programmer specifies the intermediate result and accuracy indicator variables to be recorded and the time instants to record them. The latest recorded values of these variables are made available to the client upon premature termination of the server. The client can determine whether an imprecise result is acceptable by examining the accuracy indicators.

To illustrate how availability and fault tolerance can be increased in an imprecise system, we return to the tracking example mentioned earlier. A transient fault may cause a tracking computation to terminate prematurely and produce an imprecise result. No error recovery is needed if the result allows

the system to maintain track of the target. Similarly, as long as the imprecise result produced by a control law computation is sufficiently accurate for the controlled system to remain stable, the fault that causes the computation to terminate prematurely can be tolerated. For these time-critical, embedded applications, there is often no time to take any error recovery action, and the limitation in system resources does not allow fully replicated data storage and task executions. The provision and use of imprecise results may be the only effective way to increase availability and provide fault tolerance. When an imprecise result is not usable and error recovery is necessary, the result, together with the recorded values of the accuracy indicator variables can serve as a checkpoint [34-37]. By making effective use of the semantics of the imprecise result and the computation, it is often possible to reduce the size of the checkpoint, as well as the time required for error recovery. We will return to discuss this possibility for several application domains.

In general, the imprecise computation technique can be used to increase the resilience of any operation that is not required to be atomic. For example, a bubble sort server can be made more resilient if the client invoking it can use partially sorted data. The number of passes completed by the sorter is an accuracy indicator. Whenever the sorter terminates prematurely, the value of this indicator will allow the client to decide if the array elements are sufficiently ordered. After n passes, the smallest (or largest) element is surely in the last (or first) n entries of the array. This imprecise result may be good enough if a small degree of irregularity in the output data is acceptable. An overloaded system may choose not to carry out error recovery action in this case.

The imprecise computation technique can also be incorporated with the n-modular redundancy technique. When it is possible to vote based on imprecise results, more errors (for example, errors that occur after the mandatory subtasks have been completed) can be masked. The use of monotone imprecise computations allows partial replication. Since the quality of the result returned by a monotone task always improves, we can choose to replicate only the mandatory portion of each task. After a vote based on the results produced by the replicated mandatory subtasks has been taken, only one optional subtask needs to continue. The imprecise result accepted by the voter can be used to check the precise result produced by the optional subtask. In this way, the overhead in replication can be reduced.

The above examples illustrate how imprecise computations can be used to minimize the need for error recovery and to enhance the resilience of critical computations. Again, this technique is highly application-domain specific. The imprecise systems architecture described in the next section intends to provide a framework that ensures the efficient generation and correct usage of imprecise results. We will also need domain-specific mechanisms that fit and complete the framework. They are discussed in Section IV.

III. AN IMPRECISE SYSTEM ARCHITECTURE

We are developing an imprecise system architecture for fault tolerance and dependability. This architecture includes the following key components:

(1) a *result saving protocol* — Intermediate results of monotonically improving quality must be recorded and imprecise results made available. This protocol governs the behavior of the result saving processes.

(2) an *imprecise service negotiation protocol* — This protocol governs the interactions between the application tasks and the underlying system.

(3) *error recovery methods* — The architecture should support a family of error recovery methods that make use of imprecise results.

(4) *imprecision management strategies* — Different imprecise management mechanisms are needed for the different applications supported by the system. Such a mechanism enforces the rules on when imprecise results are acceptable and what error recovery operations are to be carried out when the results are not acceptable.

Result Saving and Imprecise Service Negotiation

We are examining a process structure to determine its performance in supporting different result saving protocols, imprecise service negotiation protocols and error recovery operations. This process structure is a variation of the well-known client-server model. A server type is defined for each service provided by an imprecise system. Each (instance of a) server (type) is split into a *callee* (task) and a *supervisor*. One way to implement an imprecise server is as follows: The supervisor in a server is initially blocked waiting for invocation requests from clients. A client invokes a service by sending an invocation request to the supervisor of the server. When the supervisor receives the invocation request, it initializes the callee and requests the system to schedule the callee's execution. The supervisor executes concurrently with and monitors the progress of the callee task. When the callee terminates normally, the result produced is passed to the client through the supervisor.

In the event that the callee terminates prematurely, the supervisor returns to the client the best imprecise result produced by the callee. For this purpose, the supervisor records intermediate results obtained from the callee at instances of the callee's execution. These instances are specified by the programmer, and the recorded intermediate results are monotonically improving in quality as the task execution progresses. The supervisor also records the values of those variables specified by the programmer as accuracy indicators. If the callee crashes or if it is requested to terminate prematurely, the supervisor triggers an imprecise result handler in the client and passes to it the latest recorded values of the result variables and accuracy indicators. Based on the values of accuracy indicators, the handler can decide whether the intermediate result is acceptable. If the result is acceptable, the client may use the result as it is or modify the result as desired. If the result is not acceptable, the handler can take appropriate recovery action.

The bubble sort server example illustrates the fact that only the application tasks know how accurate an approximation must be for it to be usable. On the other hand, only the underlying system has the knowledge on the demand and the availability of all resources. This knowledge is needed to support the decision on what quality of service to provide to each client and which tasks to terminate in order for an imprecise system to recover from a component failure or a transient overload. Therefore, this decision should be made cooperatively between the application tasks and the underlying support system. To support this decision, the information regarding the accuracy of any called task required by the client and the minimal accuracy achievable by the server must be exchanged between the client, the server, and the underlying system, according to an imprecise service negotiation protocol.

With the process structure described above, the information on the required accuracy can be passed by the client to the supervisor in the invocation request. This information is then passed by the supervisor to the system in its request to schedule the callee, along with such attributes of the callee as the estimated execution times of its mandatory and optimal subtasks and their memory requirements. If sufficient system resources are available, the system then informs the supervisor, and the supervisor in turn informs the client, that the invocation request can be accepted. In this way, the quality of each imprecise service can be negotiated at the invocation time.

The critical questions for which we are seeking answers are how efficient are the result saving and the imprecise service invocation processes. The result saving process must be efficient so that it will not degrade the normal system performance substantially. Similarly, the overhead incurred in the imprecise service negotiation should be low.

Error Recovery Making Use of Imprecise Results

When a server terminates prematurely due to a crash, an imprecise result will be returned. Error recovery is only necessary when the imprecise result is not acceptable. In this way, the imprecise computation approach reduces the need for error recovery. When the imprecise results are not acceptable and recovery operations are necessary, both backward and forward recovery [34] can be effectively supported in an imprecise system.

Specifically, intermediate results of each computation are saved at appropriate instants; this result saving process can be combined with the checkpointing process. By choosing an appropriate set of accuracy indicator variables, the values of these variables returned by a prematurely terminated task can provide not only the information on whether the imprecise result is acceptable, but also a snapshot of a valid past state of the computation. In other words, the imprecise result and the accuracy indicators constitute a checkpoint. Similarly, it is possible to choose the imprecise result and accuracy indicator variables of all the computations in an imprecise system so that the values of these variables returned by the system when it crashes constitute a system checkpoint.

To take full advantage of the imprecise computation approach, the checkpointing mechanism and the recovery action should exploit the semantics of data, operations and applications. We call the approach to checkpointing, that uses the semantic information to determine what should be included in a checkpoint, the semantical approach. A complete checkpoint usually includes more information than the minimal required. Some of the saved information may be redundant. We believe that by making use of the semantic meanings of the imprecise results, it is possible reduce the size of the checkpoint and the cost of processing partial state information. We are developing a semantical approach for several key application domains.

An approach to forward error recovery in an imprecise system relies on the sequence of imprecise results returned during the course of a computation. Rather than only the final imprecise result produced at the time of its premature termination, the supervisor can provide the history of the crashed computation. The client may be able to predict the final result based on the history. This capability can be very useful in real-time systems. In such systems, forward recovery is often necessary because of rigid timing constraints. Moreover, the feasibility of predicting the final answer is often likely. A series of

imprecise results produced by a real-time computation is often the consecutive discrete time samples of a continuous time function. The future values of such a function can be estimated by extrapolating from the available past samples, that is, the returned imprecise results during the course of the task execution. For example, statistical communication theory and optimal control theory give us the rigorous theoretical foundation for optimal filters with prediction and pure prediction filters. Such a filter would receive the imprecise results, store them, and if the task terminates prematurely, apply the user's algorithm to generate a better approximation of the final result. Depending on the semantics of the problem, there may be several ways to do this extrapolation to arrive at different acceptable estimates of the final result. This mechanism can provide the user with better control of the quality of imprecise results.

Imprecision Management

Ideally, one would like to derive the rules defining the imprecise management policies in an imprecise system from the requirement specifications of the system. An imprecise system should continue to provide its services at some acceptable, although degraded, level when acceptable imprecise results are used in place of the desired results. These degraded levels are defined by a series of specifications of the system requirements. For example, in a simplistic way, an imprecise system can be viewed as having a set of minimal requirements and a set of enhanced requirements. In a correct implementation of the system, the minimal requirements are met whenever all the mandatory subtasks are completed and all the imprecise results are acceptable. The enhanced requirements are met when all the tasks are executed to completion. The criteria for accepting imprecise results can then be derived from these requirement specifications.

Unfortunately, this simplistic view is inadequate for many reasons. The most obvious one is that it neglects the following crucial facts. Faults may cause tasks to terminate at different points of their executions, providing many combinations of imprecise results with differing qualities. Imprecise results can be correctly used as approximations of precise results only when the approximations have exact, well-defined semantic meanings, there are usable metrics with which the qualities of different approximations can be compared, and the effects of inaccuracy can be rigorously bounded. Moreover, monotone computational algorithms are needed to support the chosen approximation semantics and quality metrics.

IV. DOMAIN-SPECIFIC MECHANISMS

This section discusses the considerations that must be taken into account in imprecision management for several key applications. The application domains chosen to be considered here include numerical computations, signal processing, control, and database queries and updates. A reason for choosing these domains is that they are commonly encountered in time-critical, embedded systems. In addition, managing imprecision in these domains presents us with a full range of challenges.

Numerical Computations

We are developing a comprehensive set of imprecision management policies suited for numerical computations. Again, an imprecision management policy defines when an imprecise result is acceptable and usable as it is, whether to carry out error recovery, and what error recovery operation to carry out. It ensures the correct usage of imprecise results. We believe that the insight we will gain from this

important and much studied domain will give us a better handle on managing imprecision in other application domains, such as symbolic computations.

A problem to be solved by a numerical computation can be characterized in general using the equation $f(x,t)=0$, where the answer to be computed is x and the problem f changes with time t. In most embedded, real-time applications, the answer to the problem is needed at a sequence of time points which may or may not be periodic. It is impossible to compute an exact answer in finite time, even in the absence of faults, for two reasons: finite precision and finite dimensional approximations to infinite-dimensional problems. Algorithms for solving these problems typically provide more accuracy if larger amounts of computation time are used. Therefore, computations are usually structured so that when they terminate normally, their exact results have the desired accuracy. The remaining numerical error in an exact result is known to be tolerable; in our discussions, we say that the accuracy of an exact result is one.

In an imprecise computation system, we allow computations to terminate prematurely and produce results with less than the desired, and possibly variable, degrees of accuracy. The effect and the propagation of such numerical errors need to be re-examined. For the purpose of this study, we need to distinguish three different types of problems: Types 1, 2, and 3. Different imprecise management policies will be needed for different types of problems.

In the case of *Type 1*, the simpliest kind, the problem to be solved at the current time point is unrelated to any previous problem solved or is so different that the previous solutions are of no value. No information can be carried forward from one time point to the next. The decision on whether to accept an imprecise result produced by a prematurely terminated Type 1 computation can be made independently of previous results. When the system recovers from a fault causing a premature termination and there is time, it is clear that error recovery action should be carried out to improve the result as much as possible. The problem is simply that of deciding how to allocate system resources among the computations competing for them. This problem is essentially one of scheduling.

A *Type 2* problem to be solved at the current time point is similar to a problem solved at a previous point with small changes due to small changes in the input data. However, the solution does not depend on the solution at that previous time point. An example is a traditional control problem. The solution is based on the latest values of the state variables and input variables that characterize the problem. Information from the previous solution can be carried forward to the current time point. In particular, the solution of the previous time point can be used to provide an initial approximation to the solution at the current time point, possibly even one accurate enough for use if there is insufficient time to do further computation. The decision on whether to accept an imprecise result at the current time point can no longer be made independently of previous results as in Type 1 problems. When the system recovers from a failure, it is no longer clear what to do: to do error recovery and improve the imprecise result even though the result is sufficiently accurate, or to accept the imprecise result and use the available processor time to do some preliminary calculation for future time points. To make the optimal choice in this case will require the use of a cost function that includes the accuracies of results at several time points. The rules used to decide whether to accept an imprecise result are likely to be much more complex than the rules needed for Type 1 problems.

A *Type 3* problem to be solved at the next time point is a continuation of a problem solved at a previous time point. Its solution depends on the solutions at all past time points. An example of this type of problem is the continuing solution of a differential equation in real-time simulation. The information from previous solutions is needed to define the solution at the current time point, and it may also be used to provide an initial guess to the solution at this time point. Again, it is not so obvious how best to use the system resources after an acceptable imprecise result has been returned. In some situations it might be preferable to roll back to compute the solution of a previous time point to increase its accuracy, since the numerical error remaining there will propagate into the current and future time points. Type 3 problems present a much greater challenge. The cost function will have to involve the numerical errors at all time points. In addition to the considerations of Type 2 problems, the dependency of future numerical errors on current numerical errors must be considered.

Key Application Domains and Multiple Domains

Image and voice transmission, signal processing, and control are classes of applications for which approximate results have different semantic meanings. Signal processing and traditional feedback control represent two different types of application: error-noncumulative and error-cumulative [17,18,25]. Intelligent control systems are typically non-linear. A small error in the control input can lead the system to a final state far from the desired one. These applications will need different imprecision management strategies based on different considerations.

Image and Voice Transmissions

Still image transmission is a practical example of Type 1 problems, except for the fact that the tasks to be done are data transmissions instead of numerical computations. Image transmissions are called error-noncumulative tasks [17,18,25]. The effects of inaccuracy in the results produced by different error-noncumulative tasks are not cumulative. For example, suppose that a progressive built-up method is used for the transmission of still images [38]. The data encoding each image frame is divided into four blocks; each additional block gives a clearer image. If the transmission of any image terminates prematurely due to network partition or congestion, the imprecise result is a fuzzy picture as long as the transmission of one of the four blocks is completed. We can decide whether the fuzzy picture is acceptable without considering the results of other transmissions. In this case, error recovery is also relatively straightforward. For this application, the protocols for both result saving and imprecision management should be relatively simple.

Transmission of compressed video (or voice) is a more complex error-noncumulative application. Typically, frames of images are transmitted periodically. When the transmissions of some frames are terminated prematurely, only the average inaccuracy in incompletely transmitted frames in several consecutive periods has an observable effect. It is not necessary to complete the transmission of any frame. However, a sufficient number of frames of good quality must be transmitted every second in order to maintain an overall acceptable average quality. The decision on whether to accept an imprecise result, that is an incomplete frame, must be made based on the average accuracy over several periods. The fact that the timely delivery of each frame is more critical than the quality of each individual frame is also a critical factor in this decision.

Signal Processing

Signal processing is a class of applications that integrates many different types of computations. This mixture of different computations with different approximation semantics and quality metrics presents us with a real challenge in reasoning about the monotonicity of the overall system operation and ensuring the correct usage of imprecise results. This is true even for the simplest of signal processing problems. For example, a radar signal processor processes each radar return signal to find its frequency spectrum, estimates the power spectral density of the background noise, carries out statistical detection to generate track records, correlate individual track records, etc. More complex signal processing problems may require computations to do feature extraction, pattern matching, and object identification. The imprecise results generated in these steps have different semantic meanings and different quality metrics. In other words, signal processing applications are typically of multiple domains.

Fortunately, signal processing tasks are typically functional, and reasoning about the monotone property of multiple-domain tasks that are pure functions is simpler than for tasks that maintain states between invocations. A multiple-domain functional task may be implemented by several primitive operations; all of these operations must all be functional. For example, the overall task of a simple radar signal processor consists of primitive operations such as FFT, noise estimation and threshold detection. To show that a functional multiple-domain task is monotone (or is not), it suffices to show that each of the primitive operations is monotone (or some of them are not). The fact that its operands are from different domains and have different semantics will require new and useful quality metrics for comparison. These metrics need to be developed.

Similarly, we will need guidelines and rules for determining when an imprecise result produced in an individual step is sufficiently accurate, and how to bound the inaccuracy in the final result produced by the system as a whole. At this time, we do not understand all the relevant issues well enough to find general solutions. By studying the multiple-domain imprecise management problem encountered in signal processing and by seeking specific solutions for this type of application, we hope to gain more insights into the general multiple-domain problem.

Control Systems

A traditional linear or approximately piece-wise linear feedback control is an example of Type 2 problems. Computations of control laws are called error-cumulative tasks [17,18,25]. Such a task is typically executed periodically. An answer is needed each period. The answer produced in the previous period can be used as an initial approximation of the answer in the current period. Because the effects of inaccuracy in the answers cumulate over multiple periods, the decision on whether to accept an imprecise result in the current period must be made based on the quality of the results produced in several consecutive periods. In this case, imprecision management rules should follow from control theoretical requirements on overall system stability and controllability.

Managing imprecision for non-linear control systems (such as bang-bang control) and hybrid systems (such as knowledge-based control and discrete-event/continuous control) is more difficult. Often, a small error in the control decision can lead the system to a state far from the desired final state. For example, in a bang-bang control system, the input excitation is switched between its maximum allowed

positive and negative values in order to bring the system to a desired final state in the shortest time. Each time, the decision on when to switch the excitation is based on the computed system trajectory in the phase plane of the system. A small error in the time (or the system state) to switch the control input can put the system on an erroneous trajectory. This trajectory can either lead to a large error in the final state reached by the system or significantly lengthen the time required to bring the system to the correct final state. The decision on when an imprecise result is acceptable may depend on all the past results, that is, the entire trajectory and adjacent trajectories. The considerations that must be taken into account are similar to those of Type 3 numerical computation problems.

Database Queries and Updates

Almost every computing system contains a database of some form. Traditionally, database query (and update) operations are atomic; an answer to a query is returned only after all the data required to answer the query are retrieved and processed. If a failure causes some of these data to become unavailable, no answer is returned. To support imprecise computations for fault tolerance, query (and update) computations must be restructured to provide useful approximations of precise answers based on partially retrieved data.

The problem of providing approximate answers to database queries has been approached from several different perspectives. Most of the efforts by the database and knowledge-base community have been concerned with providing useful approximate answers when the information contained in the database is incomplete or the query expression is imprecise. Approximate answers to database queries can be viewed as vague or similar answers [39-41]. If an answer that precisely matches the query qualifications is not available, an answer closely matching the qualifications is returned as the response to the query. Other efforts have developed models to deal with partial values in relational databases and theories for processing incomplete information (for example [42-45]). In contrast, the use of imprecise answers for fault tolerance purposes requires that approximate answers of monotonically improving quality be returned as more data are retrieved and processed.

Monotone Processing of Set-Valued Queries

Approximate answers of monotonically improving quality can be easily provided in statistical databases as demonstrated by the work described in [46]. We have recently developed a monotone query processing algorithm that produces improving approximate answers to queries posed in any standard relational algebra language [47,48]. This work assumes that the information in the database is complete and the queries are precise. The imprecision in the answers arises because the data required to answer the query have not been completely retrieved and processed. An answer to such a query is set-valued; it is a relation, a set of tuples. For a set-valued query, a meaningful and useful set of approximate answers can be defined in terms of subsets and supersets of the precise answer, as suggested in [49]. This semantics of approximation can be formally captured by the approximate relational model. This model defines the approximations of any standard relation in terms of supersets and subsets of the relation, a partial-order relation over the set of all approximate relations for comparing them, and a complete set of approximate relational algebra operations as primitives of monotone query processing. Here, by a *standard relation* (or simply a relation), we mean a relation in the traditional sense.

To see the general applicability of the approximation semantics defined by the approximate relational model for all set-valued queries, we note that subsets and supersets of a relation can be used to form approximations of the relation. A superset of a relation S contains all the tuples that can possibly be in S, and a subset of S certainly contains no tuples that are not in S. Let υ be the set of cartesian products of all domains in the schema of a relation S. υ is the set of all tuples that could possibly be in S. For a given relation S, we partition the set of all tuples in υ into three blocks: tuples that are certainly in S, tuples certainly not in S, and tuples whose membership in S are uncertain. We define an approximate relation of S in terms of subsets of tuples from two of these blocks, those that are certainly in S and those whose membership in S is uncertain. Formally, an *approximate relation* R of a standard relation S is a superset of S that is partitioned into two blocks, C and P, called the *certain subset* and the *possible subset*, respectively. C contains tuples with *certain* membership in S. P contains tuples with *possible* membership in S; tuples in P are not known to be in S and are not known to not be in S. The approximate relation R is denoted by the 2-tuple (C, P). (The subset of all tuples certainly not in S is $\upsilon - C - P$.)

A standard relation S has many approximate relations. Given a set of all approximations of S, we define a partial order relation $\geq$ over the set as follows: the approximate relation $R_i = (C_i, P_i)$ is said to be *better* than or equal to the approximate relation $R_j = (C_j, P_j)$, denoted as $R_i \geq R_j$, if (1) $C_i \supseteq C_j$, (2) $P_i \subseteq P_j$, and (3) $C_i - C_j$ is a subset of the tuples in P_j. In other words, as the approximation improves from R_j to R_i, tuples in $C_i - C_j$ have migrated from the possible subset of R_j into the certain subset of R_i because their membership in S becomes certain. Some other tuples in P_j have been discarded because their not being in S is ascertained. It has been shown that the partial ordered set of all approximations of S is a lattice [47,48]. In this lattice, $R_0 = (\varnothing, \upsilon)$ is the least (that is, the worst) element, where $\varnothing$ denotes the empty set. $R_0 \leq R_i$ for all approximate relations R_i of S. The greatest element, that is the best possible approximation, is S itself, which can be represented by $(S, \varnothing)$.

The approximate relational model also defines a complete set of approximate relational algebra operations [47,48]. Each operation accepts approximate relation(s) as operand(s) and produces an approximate relation as its answer. Every operation is monotone: the result of a monotone relational algebra operation becomes better as its operand(s) becomes better.

We have designed a monotone algorithm for processing relational algebra queries using the approximate relational algebra operations as primitives. This algorithm begins by computing from the scheme of the precise answer S an initial approximation of S. (When υ is finite, $R_0 = (\varnothing, \upsilon)$ is a possible, but the poorest, initial approximation.) As more base relation data are retrieved and processed, the membership of more tuples in S is ascertained. Consequently, some tuples in the possible subset migrate to the certain subset, while some others in the possible subset are discarded. In this way, this algorithm produces intermediate answers that improve monotonically as more data are retrieved and processed. Each approximate answer integrates the effect of additional retrieved data. None but the final, precise answer requires the read requests for all the base relations be completed before it can be produced. The partial-order set of all approximations of the precise answer being a lattice ensures us that the final answer is the precise answer. If query processing is prematurely terminated due to a fault or network partition, the best approximate answer produced at the time is returned.

A prototype monotone query processor has been implemented. It is designed to provide a standard relational algebra query interface to a standard relational database system. The query processor has an interactive user interface which allows the user to stop query processing at any time. The query processor, upon receiving a stop request from the user, returns the approximate answer generated at the time when the request is received. The user can request query processing be resumed after viewing the approximate answer.

Approximation Semantics for Single-Valued Queries

Database queries are often single valued. A single-valued query expresses some properties of the object of interest as a whole, and the expected answer may assume any of a countable set of values. Examples of single-valued queries include (1) "what is the color of the car no. 7826?", (2) "is the missile going to hit the large object?", and (3) "what is the wind speed over the airfield?". For these queries, the semantics of approximation given by the approximate relational model is not meaningful. Currently, we are developing an extension of the approximate relational model to support semantics of approximation suited for many kinds of single-valued queries. There are many aspects of this problem to consider.

An important aspect to be concerned with is the metric of quality for measuring the closeness of an approximate answer to the precise answer. A straightforward approach is to define a distance function (that is, a non-negative, idempotent, reflexive, and transitive function) for each database attribute and the corresponding domain of values. The quality of an approximate answer can then be measured according to the distance function defined for the domain in which the precise answer assumes its value. A simple distance function often gives us a metric of quality that is sufficient for many queries of practical interest. In example (1), suppose that the distance function chosen for a domain of colors is the difference between color spectra. Then, "purple" is a better approximate answer than "red" when the precise answer should be "silver blue". On the other hand, if the difference between shades of color is of interest, then "pink" is a better approximate answer than "navy". By choosing an appropriate distance function, we can have a useful metric of quality for either one of these semantic meanings. An answer of "yes" or "no" to the query in example (2) listed above can be based on the difference between the projected coordinates of the missile and the approximate coordinates of the center of gravity of the large object. The single-valued answer in this example is in fact based on a set of values: the set of coordinates of all the parts of the large object from which the coordinates of its center of gravity are computed. A possible measure of quality of a single value as an approximation of a set of values is its average distance, or weighted average distance, to the individual values in the set. The answers to some queries are vectors of values. In example (3), the answer is a 2-dimensional vector: the wind direction and absolute speed. To measure the closeness between two vector-valued answers in general, we need to define the distance between pairs of values from multiple attributes. By giving the distance function different semantics, for a given set of vector values and a particular value in the set, we can define either a linear order or a partial order relation over the set.

We are developing the needed semantic views of the database to support monotone query processing when the approximate answers are measured in terms of the different distance functions. We are also investigating the use of an object-oriented model as a logical model for the external view rather than confining its use to the underlying support. Because it is capable of modeling complex relationships

between data and the rich semantic information it contains, an object-oriented model will allow us to represent and characterize a wider class of approximations. Unlike the relational model, the object-oriented model lacks a rigorous theoretical foundation and a set of primitive operations that is known to be complete. Proving monotonicity in the object-oriented framework will be more difficult.

Approximate Updates of Databases

A natural extension of the approximate database query processor is to consider the case when the database contains incomplete information or partial values. This problem is closely related to the imprecise update problem. By an imprecise update, we mean a partial update of the database due to a premature termination of a complete, precise update operation.

To ensure that the database will not be in an inconsistent, unsafe state, update operations are typically atomic. A naive approach to support imprecise updates is to implement all update operations so that they are carried out in multiple phases, as they typically are in distributed database systems. A multi-phase update operation can be thought of as an atomic action that is implemented with multiple versions. When such an operation terminates prematurely, it is considered to have failed when the number of phases completed at the time is insufficient. The intermediate result produced is unacceptable. Otherwise, a multiple-version atomic action is completed. The more phases completed before the operation terminates, the better is the version. In this way, we can view such update operations as being monotone and capable of returning imprecise results. In practice, however, every partially completed multi-phase update operation must be eventually completed, after the faulty condition causing it to terminate prematurely is cleared. In other words, an imprecise result is almost never usable as it is; some form of recovery action is always needed. This approach to structure update operations does not offer us any advantage over the traditional checkpoint/recovery approach. In fact, the two approaches are essential the same.

We are investigating the feasibility of partial updates that requires no error recovery action to complete the update. Such an update may introduce uncertainty or incompleteness in the information contained in the database, but will not lead the database to an unsafe and, hence, unacceptable state. A promising approach is based on the model-based automatic update method [45]. The model-based approach has been applied to handle updates of knowledge bases that arise as one reasons about the effects on the world of any action taken to change the state of the world. An automatic update operation leads the knowledge base to one of many possible final states. The model-based approach begins with a given initial model that represents the initial state of the knowledge base. For a given update, a set of result models is computed. The result models represent the possible final states of the knowledge base when the update operation completes. The result of an update operation is acceptable as long as the operation leads the knowledge base to one of the acceptable final states.

Although the model-based approach was proposed to solve a problem quite different from our problem of supporting imprecise updates, it appears to provide us with a fruitful way to look at our problem. Suppose that every update operation is associated with a set of result models, each of which gives an acceptable result state of the database after the update. When a prematurely terminated update operation puts the database in a result state given by one of these models, its result is acceptable. Otherwise, the imprecise result is not acceptable; a recovery action is required. The work on automatic

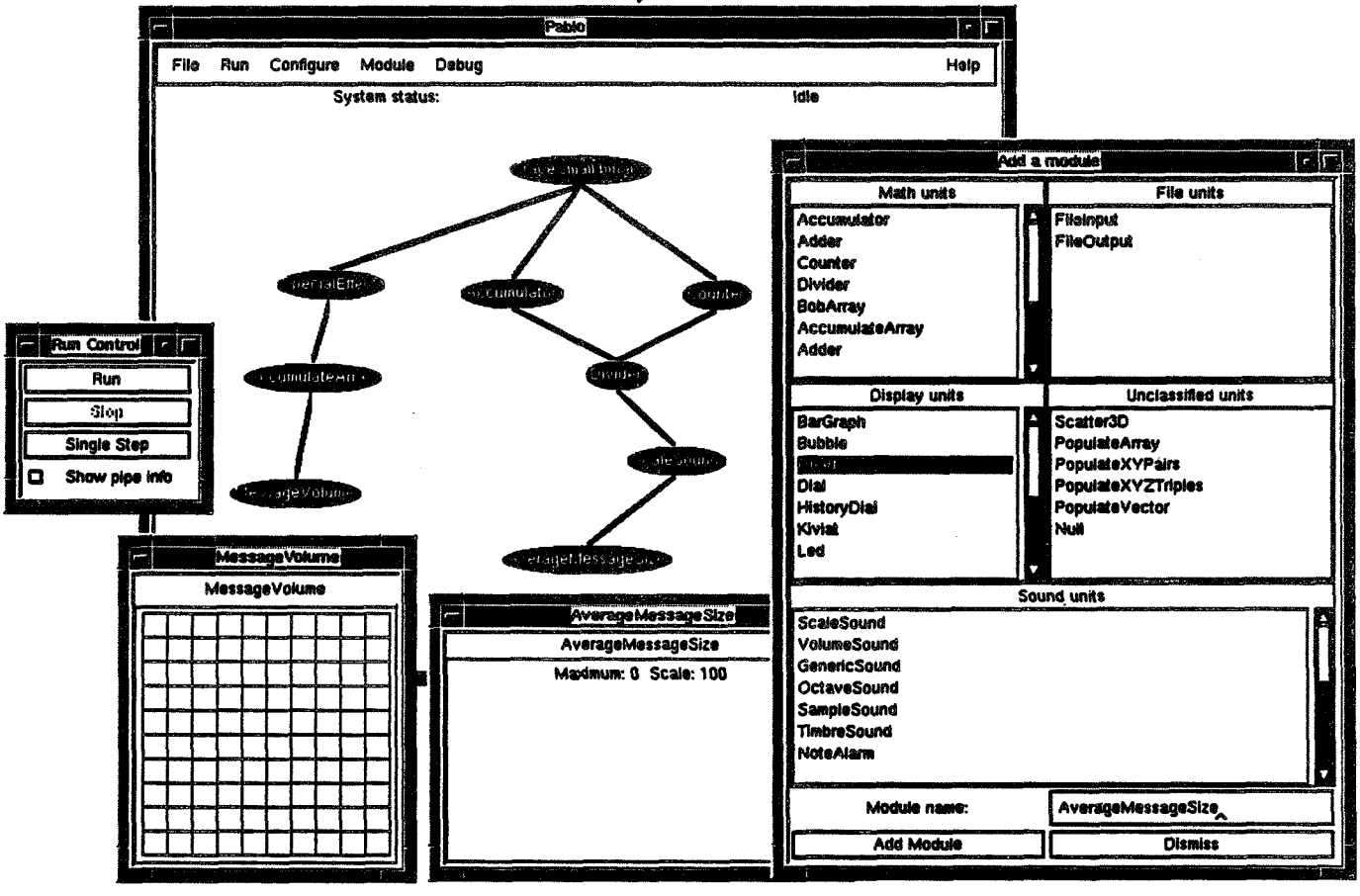


Figure 2: Pablo Module Selection

Acknowledgments

Designing and implementing a scalable performance environments is a non-trivial task. The work summarized here is the product of a many people's ideas, time, and talent. In addition to the authors, Jim Arendt, Al Malony, and Ted Nelson all contributed to the design of the Pablo environment. Finally, Sever Tipei of the Department of Music offered several valuable music and sonification insights.

References

[1] BECKER, A. Modern Virtual Screen Technology and Applications. In *Proceedings of Spring CompCon '90* (1990), IEEE Computer Society, pp. 612–615.

[2] BLATTNER, M. M., SUMIKAWA, D. A., AND GREENBERG, R. M. Earcons and Icons: Their Structure and Common Design Principles. *Human-Computer Interaction 4* (1989), 11–44.

[3] CHUNG, J. C., HARRIS, M., BROOKS, F. P., FUCHS, H., KELLEY, M. T., HUGHES, J., OUH-YOUNG, M., CHEUNG, C., HOLLOWAY, R. L., AND PIQUE, M. Exploring Virtual Worlds with Head-Mounted Displays. In *Three-Dimensional Visualization and Display Technologies, Volume 1083* (1989), SPIE, pp. 42–52.

[4] COUCH, A. L., AND KRUMME, D. W. Monitoring Parallel Executions in Real Time. In *Proceedings of the Fifth Distributed Memory Computing Conference* (Apr. 1990), pp. 1187–1206.

[5] DYER, D. S. A Dataflow Toolkit for Visualization. *IEEE Computer* (July 1990), 60–69.

[6] FRANCIONI, J. M., ALBRIGHT, L., AND JACKSON, J. A. The Sounds of Parallel Programs. In *Proceedings of the Sixth Distributed Memory Computing Conference* (1991), IEEE Computer Society.

[7] LILLEVIK, S. L. Touchstone Program Overview. In *Proceedings of the Fifth Distributed Memory*

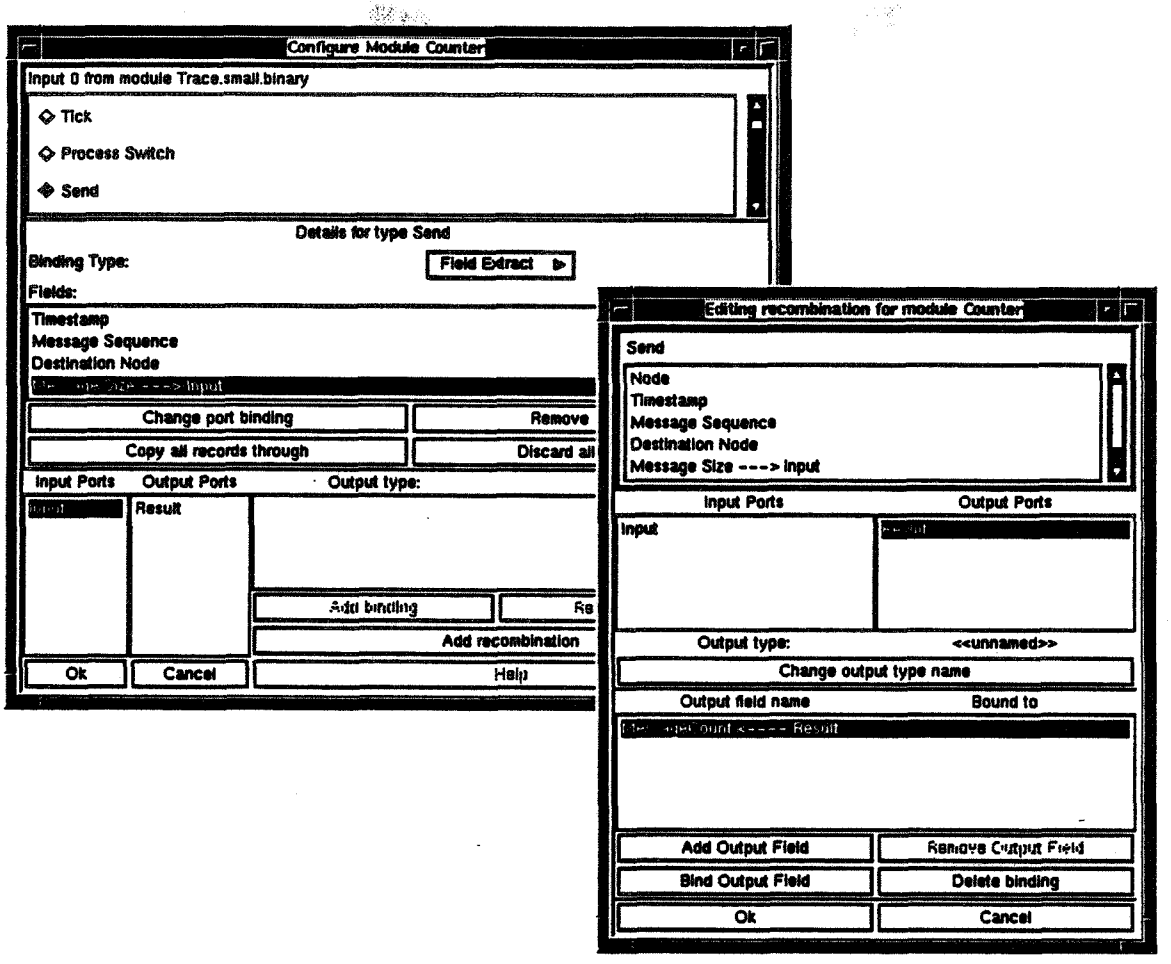


Figure 3: Pablo Data Binding

Computing Conference (1990), IEEE Computer Society, pp. 647–657.

[8] MALONY, A. D., AND REED, D. A. Visualizing Parallel Computer System Performance. In *Instrumentation for Future Parallel Computing Systems*, M. Simmons, R. Koskela, and I. Bucher, Eds. Addison-Wesley Publishing Company, 1989.

[9] MALONY, A. D., AND REED, D. A. A Hardware-Based Performance Monitor for the Intel iPSC/2 Hypercube. In *1990 ACM International Conference on Supercomputing* (June 1990), Association for Computing Machinery.

[10] MALONY, A. D., REED, D. A., AND RUDOLPH, D. C. Integrating Performance Data Collection, Analysis, and Visualization. In *Parallel Computer Systems: Performance Instrumentation and Visualization*, M. Simmons, R. Koskela, and I. Bucher, Eds. Addison-Wesley Publishing Company, 1990.

[11] MICHAEL BERRY, E. A. The Perfect Club Benchmarks: Performance Evaluation of Supercomputers. *The International Journal of Supercomputer Applications 3*, 3 (1989), 5–40. Fall.

[12] NCSA. *NCSA HDF, Version 2.0*, Feb. 1989.

[13] REED, D. A., AND RUDOLPH, D. C. Experiences with Hypercube Operating System Instrumentation. *International Journal of High-Speed Computing* (Dec. 1989), 517–542.

[14] REW, R. K. *netCDF User's Guide, Version 1.0.* Unidata Program Center, University Corporation for Atmospheric Research, Apr. 1989.

[15] STARDENT COMPUTER, INC. *Application Visualization System, User's Guide*, 1989.

[16] WILLIAMS, C., AND RASURE, J. A Visual Language for Image Processing. In *IEEE Workshop on Visual Languages* (Oct. 1990).

we plan to exploit audio's potential for a wide range of perceptually continuous transformations in real time.

Ideally, much of this sonic data exploration would be conducted interactively, with real-time output of high-quality digital sound. However, real-time interaction must be balanced against sonic complexity and available hardware support. Our current implementation uses the Sun SparcStation audio port for sound output. This hardware is ubiquitous and supports a large user base; unfortunately, the sound fidelity is very low.

The music world has adopted the Musical Instrument Digital Interface (MIDI) as a standard, and there are wide variety of electronic musical instruments with MIDI interfaces. By connecting one or more of these instruments to a workstation, one can easily create a broad spectrum of sounds [6]. For this reason, we are now extending our sonic implementation to support both the basic SparcStation audio and one or more inexpensive MIDI musical instruments.

Clearly, the practical utility of sonic output remains to be empirically determined. Its major virtue is that it can be used to convey warnings or a qualitative sense of system behavior without forcing the performance analyst to continually watch his or her workstation display.

4 Environment Examples

To illustrate the benefits of a self-documenting trace format and the operation of the data analysis environment, we describe the steps a user might follow when creating a data analysis graph to reduce a trace using this environment. As an example, Figure 2 shows a data analysis graph being configured to analyze performance data from a distributed memory parallel system.

To begin a performance data analysis session, one first interactively selects analysis modules from the analysis module palette and "wires" them together to create a new data analysis graph. Alternatively, one can load a previously created analysis graph from a library of such graphs. By placing display modules at the leaves of the graph, the user can display and interact with the output of the analysis via dynamic graphics or sound. The format of the data that flows between the modules is that of the self-documenting trace format.[2] In Figure 2, the *Add a Module* dialog allows the user to choose among file input/output, data transformation, data assembly, graphical display, and sound modules. After assigning the selected module a unique name, its icon can be interactively placed on the *Pablo* analysis module icon canvas and connected with other analysis modules.

In Figure 2, the user has created an analysis graph that will display both the running average size of all messages, represented by the *AverageMessageSize* icon and the associated strip chart, and the spatial distribution of messages sent among all nodes, represented by the *MessageVolume* icon and associated matrix display. The latter shows message source and destination nodes along the two axes; the total number of bytes sent between that node pair is indicated by a color-coded matrix entry. In the figure, the *Accumulator* module will sum the sizes of all messages, and the *Counter* module will maintain a running count of the number of messages processed. The *Divider* computes the ratio of these two values. Similarly, the *AccumulateArray* module will construct a message volume matrix from the individual message sizes. Finally, the *ScaleSound* module uses a non-linear musical scale to map the time varying average message size to a sequence of musical notes; these notes represent the same data as is displayed by the strip chart.

After connecting the desired modules, but before the analysis graph begins to execute, the user must specify those records and record fields needed by each analysis module. Why? Each module may need only a subset of the data fields in some of the trace records (e.g., one might wish to find the maximum value of certain data fields in records of a particular type). In Figure 3, the user is configuring the *Counter* module by extracting all message *Send* records from the data stream. He or she has selected the *Message Size* field in these records and has bound this field to the input port of the *Counter* module. Similarly, the record producted as output by the *Counter* module must be specified; this is the recombination dialog in the left of Figure 3. It is this ability to interactively bind fields from data records that allows the environment to process *any* trace from *any* source that satisfies the self-documenting trace format. This is the source of the environment's portability.

Finally, after configuring the data analysis and display graph, the user initiates its execution. During execution, each analysis module accepts a stream of input data, applies its transformation, and produces output for its descendent modules. Because the analysis modules communicate solely via message passing, individual modules can execute in parallel; in addition, intramodule parallelism potentially can be exploited with lightweight tasking primitives.

5 Summary

The complexity of parallel systems makes *a priori* performance prediction difficult and experimental performance analysis crucial. A performance data analysis and data presentation environment is needed to access the effects of architectural and system software variations. The environment described here is an ongoing effort. As such, it incorporates lessons learned from two previous generations of successful performance data analysis software.

Although an environment prototype is now operational, much work remains to be completed, including portions of each of the environment infrastructure, the sonic widgets, and the head-mounted display.

[2] As a consequence, the output of any module can be saved in a file for later processing.

individual data fields. Our reasons for this choice are two-fold. First, a basic set of analysis modules should be applicable to a wide range of architectures. By restricting the set of basic transformations to those realizable without knowledge of data semantics (.e.g., sorting data without knowing its meaning), we insure that these analysis modules are applicable to data from any source. Second, we hope and expect that users will develop sets of analysis modules appropriate for their architecture and problem domain.

3 Data Presentation

Experience has shown that scientific data visualization is experimental; rarely can the most appropriate visualization idiom be predicted in advance. Instead, the scientist seeks false color mappings and abstractions that emphasize those components of the data that are of most interest; these choices often are discipline dependent. Based on this experience, we believe any environment for the investigation of data presentation alternatives must be flexible, allowing comparison of competing presentation methods (e.g., visual versus sonic). Thus, we are exploring the use of head-mounted displays and sonic data presentation in addition to the traditional use of visual techniques.

3.1 Dynamic Graphics

The diversity of performance data demands an equally rich set of performance displays. In our earlier performance environment, described in [10], we developed a variety of X windows performance data displays, including dials, bar charts, LEDs, Kiviat diagrams, matrix views, X-Y plots, contour plots, strip charts, and a general purpose graph display. Based on the widget paradigm, each display is independent of data semantics and supports a set of *resources* and *methods*. Display resources are the configuration parameters used to define a display instance; the display methods control the display's operation. Further, each display supports standard interface conventions; this promotes ease of use and rapid construction of new displays.

Despite the existence of a flexible set of graphic displays, we believe some fundamentally new techniques will be needed to analyse and present performance data from massively parallel systems — displays appropriate for ten processors often do not scale to a thousand processors. Fortunately, when an application program executes on a parallel system with hundreds or thousands of processors, rarely, if ever, are there more than a small number of distinct behaviors (i.e., the behavior of most processors can be grouped into a small number of equivalence classes). To support data presentation for massively parallel systems, we plan to extend our existing displays to support interactive clustering of performance data from multiple processors (i.e., data display zoom and pan), and a modest number of new displays will be created. As an initial step, we have created a three-dimensional scatter plot display that allows the user to interactively change display scales and perspectives; we are now augmenting this display to support stereopsis.

3.2 Head-Mounted Displays

One of the disadvantages of standard scientific visualization techniques is that they require the user to remain at a workstation. In addition, they often consume most or all of the screen real estate, occluding other work. Performance instrumentation can take many forms, including simply monitoring a computation until it reaches a state where more detailed instrumentation is appropriate. Head-mounted displays provide a non-intrusive alternative to visual monitoring on a workstation.

Stereoscopic, head-mounted displays with real-time or near real-time graphics often are associated with virtual reality research [3]. With current technology, these displays are heavy, awkward, and confining. However, lightweight, low-bandwidth displays need not occlude vision nor prevent normal office activities.

Reflection Technology has developed an IBM PC CGA-compatible monochrome display that weighs 2.25 ounces and occupies a package approximately 1.2"x1.3"x3.5" [1]. Images can be transferred to the display via a standard RS-232 interface. Using this display, we have developed a graphics library that supports a subset of the displays described in §3.1, albeit at reduced graphic resolution. These displays are not intended as a replacement for workstation graphics, but rather as a complement. Using this device, one can unobtrusively monitor system activity, transferring graphics to and from the workstation screen when appropriate. We are working to integrate this device with our performance analysis environment.

3.3 Sonification

The use of scientific data visualization to provide new insights is well established. However, visualization is but one way of presenting data. Psychological studies show that users more quickly respond to audio cues in certain instances. The dimensions of the audio parameter space include duration (time), pitch (frequency), volume (amplitude) and timbre (waveform). Other perceivable sound qualities include spatial location and, for music, vibrato (frequency modulation) and tremolo (amplitude modulation).

Clearly, sound represents a rich communication channel that has heretofore been highly underestimated or ignored completely in computer systems performance data presentation; the recent work by Francioni [6] is an exception. We are developing a set of "sonic widgets" analogous to the graphical widgets found in the X window system. At present, our sonic widgets support the mapping of a sequence of scalar data values to

- musical scales with different tunings,
- volumes,
- timbres, and
- arbitrary sound samples.

We have also developed a set of "earcons," [2] or audio warnings, that presently include sampled voice warnings and enumerations, alarms, and bells. Simply put,

```
/* "Instances of the actual data would
    follow this record definition."
*/;;

#1 : "MessageStatistics" {
   // "Timestamp" "Data creation time"
   long   "Timestamp";

   // "MessageVolume" "Number of bytes
         sent between each node pair"
   int    "MessageVolume"[][];

   // "MessageCount"  "Number of messages
         sent between each node pair"
   int    "MessageCount"[][];

   // "AverageSize"   "Average message size"
   double "AverageSize";

   // "MinimumSize"   "Minimum message size"
   int    "MinimumSize";

   // "MaximumSize"   "Maximum message size"
   int    "MaximumSize";

   <OTHER DATA DEFINITIONS OMITTED>
};;
```

Figure 1: Example Record Definition

editor and an analysis module selection menu, a user can interactively select analysis modules, place their icons on the icon canvas and connect (or disconnect) them to other analysis module icons.

In addition to graphically connecting modules, one must specify the behavior of each module, including what portion of the input stream is bound to the module's input ports, passed through unchanged, or discarded. Similarly, one must specify the output stream insertion point for the module's data output. A user specifies these data bindings via a graphical *Configuration Editor*. Cut-and-paste selections are used to select, modify, and map data fields in the stream input and output records. The system then configures *Data Analysis Module Wrappers* to realise the specified data manipulations when the graph begins execution. We believe this "wiring diagram" approach will allow users to quickly prototype and test simple data reductions without writing any new code.

As noted earlier, the environment has been designed to allow individual analysis modules to execute in parallel, given the necessary underlying parallel system. At present, a *Module Scheduler* insures that modules execute subject to their data dependence constraints; we will soon extend this scheduler to accommodate parallel module execution.

A *Defaults Manager* allows the user to customize the environment to his or her tastes. Configurable defaults include such system parameters as the default locations of preconfigured analysis graphs and data files and the ability to enable or disable analysis module tracing for debugging purposes. As the environment continues to grow and evolve, we expect to add additional configuration options (e.g., to specify display colormaps, to inspect module data ports, and to specify synchronous or asynchronous parallel module execution).

2.3 Data Transformation and Analysis

Although a flexible environment infrastructure is necessary, it is not sufficient. A set of commonly used data analysis modules also is needed. We are developing a set of performance data analysis modules that support simple data reductions and transformations. Our goal is not to construct a large body of analysis modules; in many areas (e.g., statistical data reduction) extant tools provide far greater functionality and generality. Instead, these modules will provide the foundation for further software development by the environment's user community — experience has taught us that we are unlikely to anticipate all potential user wants or needs. Written as C++ objects, the minimal set includes statistical analysis, data transformations, reductions, and amplifications. Examples of these might include the following, although only implementation and experience will determine a necessary set — our development is continuing.

1. statistical analysis
   - moments (mean, variance)
   - extrema (minimum, maximum)
   - skew
   - median and mode
   - frequency and percent
2. data stream transformations
   - sort
   - rank
   - scale
   - smooth
   - simple mathematical functions
   - transform (log, absolute value, etc)
   - combine streams (ratio, sum, difference)
3. data stream reductions
   - sum
   - count
   - select based on criteria
   - fit (simple, polynomial, log, interpolated)
4. data stream amplifications
   - assemble (vector, array)
   - duplicate

All modules operate in "stream" mode. Some functions (e.g., median or sum) reduce the total volume of data; others (e.g., transform) simply scale selected fields from the input stream.

Our set of analysis modules does not embody knowledge of a particular machine architecture, nor do the modules impose architecture dependent meanings on

passing, individual modules can potentially execute in parallel, allowing the construction of scalable data analysis graphs.

Each module consists of a user-written data transformation module and a system-provided data access "wrapper" that can read and write the self-documenting data format. The wrapper's data access methods permit retrieval and generation of data field types, sizes, and names without *a priori* knowledge of the data stream structure, and associated software supports data replication and distribution to other modules. The user's data analysis module need only apply the intended data transformation. Thus, one could write a function that maintains a running average without knowing the range of data values, their semantic interpretation, or their storage scheme.

This "plug-and-play" approach to performance environment infrastructure relies on four environment components:

- self-documenting data formats,
- module interconnection and configuration,
- data transformation and analysis, and
- data presentation.

Each topic is described briefly below and in §3.

2.1 Standard Trace Formats

A flexible performance environment must be capable of processing data drawn from the hardware, system software, and application programs of diverse parallel systems. This permits integration of performance data across multiple levels of a single system and cross-architecture performance comparisons using data from multiple systems.

Because performance data can take such disparate forms, it seems unlikely that one can develop a portable performance data trace format that includes embedded data semantics. However, by separating data syntax from data semantics, one can define a flexible data format capable of describing sophisticated data structures. Any data stored in this format includes an embedded description of the structure of the data.[1] Data semantics become the domain of the analyst and the performance analysis software.

The notion of self-documenting data files is not new. Users of supercomputer centers store scientific data sets and raster images in the Hierarchical Data Format (HDF) [12] or the Common Data Form (netCDF) [14]. Both formats allow the user to include a description (e.g., array sizes, types, and labels) in a file with the raw data. However, the primary intent of both HDF and netCDF is to support storage of data arrays rather than the streams of heterogeneous records typical of event traces taken from parallel systems. Because there are no equivalent, standard formats for performance data, we have defined a trace description language, a trace data meta-format.

When considering the design of a performance data trace description language, one must consider several issues, but the most important are compactness and portability. Traces can be large (many megabytes or gigabytes for highly parallel systems), and any trace definition must not preclude the possibility of data compression. ASCII traces are human-readable and highly portable, but are space inefficient for most types of data. In contrast, binary traces are compact, but byte orders and number representations differ across machines. As a compromise, we have developed both binary and ASCII trace data formats, defined by a simple grammar, and a set of software tools that support automatic conversion between formats. The existence of both binary and ASCII formats allows data migration across heterogeneous architectures. For example, to move a binary trace to a machine with a different word length or byte order (a) convert it from binary to ASCII, (b) move the ASCII version to the new machine, and (c) convert the ASCII version back to binary.

Within these constraints of portability and compactness, we have defined a trace data format that includes internal definitions of data types, sizes and names. Figure 1 shows one example of a trace record definition that one might use when representing the spatial pattern of message transmissions among the nodes of a distributed memory parallel system; fields from records of this type are used by the *Message Volume* display of Figure 2. Logically, a trace file in this format consists of a set of record definitions and a stream of record tag/data record pairs. Each tag identifies the type of the record to follow. Using this tag and the record description, the performance analyst can interactively configure the environment to select those record fields to be passed to each analysis module's input ports.

Simply put, the record definitions are equivalent to C structures or Pascal records with certain restrictions. After considerable analysis, we concluded that pointers, self-referential data types, and a hierarchy of user-defined data types would cause more implementation difficulties than the potential benefits might justify. Despite these limitations, it is still possible to define structures that contain multi-dimensional arrays of the basic machine types. In addition, only the number of array dimensions is fixed; the size of each dimension is contained in each data record.

2.2 Module Configuration

Because our performance environment is designed to accommodate diverse parallel computer systems, trace data extraction, reduction, and display must be flexible and customizable. The keys to this flexibility are a set of environment configuration options that allow the user to interactively create, configure, and execute a performance data analysis graph.

A *Graph Editor* manages an icon canvas that displays the current state of the data analysis graph. Via this

[1] Because the format can describe arbitrary data records, not just a specific record structure, it is best viewed as a meta-format.

- Detailed performance data, captured by tracing parallel program execution, is key to understanding the dynamics of system performance.
- Given raw performance data, a flexible performance data analysis *environment* is needed that includes statistical data reduction, sonification, and visualisation.
- It should be possible to *dynamically configure* data analysis and presentation components, allowing the performance analyst to change data perspectives during execution.
- An *extension interface* should permit addition of new components or modification of existing components without intimate knowledge of environment infrastructure.
- The individual analysis and data presentation components should be *easy to build* for many different application types and system software environments.
- The environment should be *fast*, preferably fast enough to process bursts of real-time data. For parallel systems, the environment performance must scale by allowing the environment's components to execute in parallel.
- The environment should support *multiple analysis levels*, including hardware, system software, and application.
- Finally, the data analysis environment should be *portable* to different systems. Although the mechanisms for performance data capture are inherently system dependent, performance data reduction, visualization, and sonification are largely system independent.

The remainder of this paper describes the components of a scalable performance analysis environment that can be adapted to a wide variety of parallel systems. In §2, we describe the environment organisation, followed by descriptions of a flexible event trace format and the individual environment components. In §3, we describe options for data presentation, and illustrate the operation of our current environment prototype in §4. Finally, §5 summarizes the current environment state and plans for future implementation.

2 Environment Organization

Designing a portable performance analysis environment capable of analysing data from massively parallel, shared or distributed memory, SIMD or MIMD systems is a daunting task. The environment must be capable of processing data from diverse sources, it must accommodate a wide variety of performance metrics, its performance and data analysis capabilities must scale with that of the systems being analyzed, and it must be capable of comparing the performance of multiple applications and architectures. Embedding knowledge of performance data semantics in performance data analysis modules precludes their reuse with both different parallel systems and different data; a module that computes sliding window averages need not, and should not, contain knowledge of data semantics (e.g., whether the data represents context switches or processor utilisation). Similarly, the idiosyncrasies of performance data formats should be invisible to data analysis modules.

Simply put, it should be possible to write most performance data analysis modules without knowledge of either data semantics or data format. Moreover, it should be possible to interconnect modules in a variety of configurations and to support parallel analysis of performance data. As the size of extant parallel systems increases, parallel processing of performance data will become necessary to provide acceptable response time. Such an environment infrastructure would permit rapid construction and testing of new analysis modules, use of multiple module instances for different purposes, and reuse of existing modules for different parallel systems.

Our prototype environment contains a set of performance data transformation modules that can be interconnected in user-specified ways. It is the responsibility of the environment infrastructure to hide details of module interconnection and data sharing. The environment is written in C++ with the graphical displays based on X windows and the Motif toolkit. It allows users to interconnect and configure modules graphically to form an acyclic, directed graph of data analysis and presentation modules. This is an extension of our previous system, described in [10], that permitted only a single level of interconnected analysis modules with fixed functionality.

Graphically connecting modules to create an executable data flow graph has been used by Couch in his Seecube performance environment [4] and in both commercial and research implementations of graphics and image processing software. The best known such example is Stardent's AVS visualization system [15]. Academic examples include the aPE visualisation system [5] and the Khoros image processing system [16]. Based on the broad success of this graphical construction paradigm in other problem domains, we believe it is an ideal approach to interactive performance data analysis.

In our environment, performance trace events are represented in a stream data format that includes internal definitions of data types, sizes, and names. This self-documenting data format is the *lingua franca* for module data sharing. Although data reduction occasionally requires knowledge of data semantics, we believe that isolating this knowledge will promote module reuse. Most modules provide semantic-independent data reductions (e.g., averaging or histogramming) on user-specified data stream components. Module output can include arbitrary combinations of the reduced data and unmodified components of the input stream; both may be used by downstream modules. Finally, because the modules communicate solely via message

2 The *Choices* Object-Oriented Operating System

Choices has, as its kernel, a dynamic collection of objects. System resources, mechanisms, and policies are represented as instances of classes that belong to a class hierarchy. For programming convenience, the root of the hierarchy is class *Object* and the classes of the hierarchy are represented at run-time as *Class* Objects.

The system has over 300 classes and 78,000 lines of source code. *Choices* runs on bare hardware: the Encore Multimax, the Apple Macintosh IIx, the AT&T WGS-386, and the IBM PS/2. It supports both uniprocessor and multiprocessor architectures.

All entities in the operating system are modeled as objects and include: system processes, user processes, regions of memory, files, and hardware devices like CPU's and disk controllers. The application/kernel interface is defined by method invocations from objects in user mode to objects in kernel mode. In user mode, a kernel object is represented by an *ObjectProxy*[10]. An ObjectProxy is created dynamically on demand if the protection policy of the kernel object is not violated.

A *Choices* Process executes in a Domain. The Domain maps a collection of logical data stores or MemoryObjects into a virtual memory[9]. The MemoryObject provides random access to the contents of the data store. A MemoryObject can be memory-mapped into virtual memory. Direct access to the contents of the MemoryObject is provided by using the virtual memory hardware mapping mechanism. Each memory-mapped MemoryObject can be paged to its own backing store using independent paging algorithms. Distributed virtual memory[7] allows the dynamic addition or removal of a MemoryObject to or from the different virtual memories of a network of computers.

Persistent data is implemented as a backing store that is loaded as a MemoryObject into virtual memory when needed. Persistent objects are encapsulated persistent data items defined by a class hierarchy. They are activated on demand and deactivated automatically[1]. Persistent objects may store references to other persistent objects.

The mechanism underlying persistent data is derived from an object-oriented file system model. The model separates access methods from implementation details. MemoryObject subclasses support access to data stored on disks, files, or primary memory. The data may be formatted in various ways including the representations used to support UNIX 4.3 BSD, MS-DOS, and UNIX System V file systems[8]. Similarly, the access methods may be specialized to accommodate application programs that are written to conform to UNIX 4.3 BSD, MS-DOS, and UNIX System V file system interfaces.

Security is provided by a combination of access rights maintained by the file system, individual objects, and a user/system protection mechanism. Kernel objects are protected by supervisor state and by the virtual memory hardware. Method calls to kernel objects from user applications cannot be performed unless the user has been granted permission to access the object. That permission is provided as an ObjectProxy. Once provided, all method calls to the ObjectProxy are trapped and converted to method calls to the kernel object. During the trap, they are checked for validity by a kernel protection mechanism. Non-kernel objects are mapped into the virtual memory space of the application. Their data segments may be shared between applications using shared virtual memory or distributed virtual memory. Once access is established, method calls can be made directly from the application to the object.

3 Automatic Deletion of Unused Objects

In *Choices* all resources, processes, data, and hardware devices are represented within an object-oriented model of the system. The system requires efficient mechanisms to create and delete objects shared by many Domains. For example, several Processes, some of which are in different Domains, may open the same file for read/write access. The file system represents the persistent data in the file as a MemoryObject. Reads and writes to the persistent data are supported by an Inode instance that is created on open and deleted on close. The Inode maps reads and writes of logical blocks into reads and writes of physical blocks of data using the BSD or System V file system block indexing algorithms. A FileStream instance provides a read/write interface to an application program and implements the UNIX notion of a sequential byte stream and current position within the file. Each application process opening a file uses a FileStream as its interface. However,

it may share the interface with other processes in the same Domain. In both cases, Inodes and FileStreams cannot be deleted until all the processes that access them have closed the file. Automatic deletion of Inodes and FileStreams when they are no longer required simplifies code and eliminates system programming errors.

Although ad hoc reference counting and other garbage collection techniques can be employed to implement the features of the operating system, these mechanisms arise so often in an operating system implementation that we have chosen to embed them within the C++ base classes that we use to support our software development. Several object-oriented languages, including Smalltalk[5], allow objects to be constructed explicitly and delete them implicitly. In C++, however, objects must be both constructed and deleted explicitly. This places a significant burden on the programmer, since in all but the most trivial object-oriented systems, it is impossible to determine at compile-time when each object should be deleted. There are two methods of determining at run-time when to delete objects: garbage collection and reference-counting[5].

3.1 Garbage Collection versus Reference-Counting

Garbage collection algorithms start with a set of objects, called *root* objects, that are by definition in use. They determine all objects that are reachable by any chain of references from the root objects, and then they delete all objects that were not reachable. Reference counting algorithms keep a count of the number of pointers to each object. When an object's reference-count reaches zero, the object is deleted.

Garbage collection is often considered expensive, though some algorithms have negligible overhead per object deleted. Even efficient algorithms, however, may have unacceptable interactive performance. Reference counting overhead is proportional to how often pointers to objects are stored.

Garbage collection algorithms for languages like Smalltalk can effectively determine all objects to delete; reference counting algorithms for most languages, however, usually fail to detect unreachable objects that belong to "pointer cycles." To implement an effective garbage collection algorithm, one must be able to determine the location of objects and which words of storage are pointers to objects[12].

We chose to use reference counting for two reasons: first, reference counting has predictable space and time overheads, and second, run-time information about the location of C++ objects and pointers is not readily available.

3.2 Reference-counting Design and Implementation

Having chosen to use reference-counting for automatic object deletion, we designed a set of classes, methods, and programming conventions to automate the process as much as possible.

All objects in the system that require automatic deletion inherit reference-counting behavior from class Object, which has an integer reference-count and five member functions related to this behavior: Object, reference, unreference, noRemainingReferences, and ~Object.

Object() The constructor for class Object initializes the object's reference-count and inserts the object in the global object table. The global object table enables storage leak detection (by displaying all objects that still exist at system shutdown) and the display of members of classes at run-time (see Section 4).

reference() The public member function reference atomically increments the object's reference-count; it must be called each time a pointer to an object is stored. To increase performance, reference is designed so that it can be made an inline function.

unreference() The public member function unreference atomically decrements the object's reference-count and calls noRemainingReferences if the object's reference-count reaches zero. It must be called each time a pointer to an object is overwritten. To increase performance, reference is designed so that it can be made an inline function.

noRemainingReferences() The protected, virtual member function noRemainingReferences(), which should only be called by unreference, calls the object's destructor by deleting "this." This function can be

overloaded to define statements to be executed *before* the object's destructor is called; no statements, however, can be executed *after* the destructor has been called.

~Object() The protected destructor, which should only be called by noRemainingReferences, removes the object from the global object table. To avoid premature deletion of objects, all subclasses of class Object must define protected destructors.

These five methods provide an effective and flexible mechanism to implement reference-counting behavior of objects, but they still place too heavy a burden on the programmer. This burden is the requirement that calls to reference and unreference functions be placed at *all* appropriate places throughout the code. Experience showed this requirement was too difficult to satisfy. To remove this burden, we chose to treat pointers to reference-counted objects as objects themselves.

3.3 Pointers as Objects

The *ObjectStar* class defines "first-class" pointers to Objects. Instances of ObjectStar have a traditional C++ pointer (`Object * _pointer`) as their only data member. Because ObjectStar defines no virtual member functions, its instances have no vtable pointer and therefore require the same amount of storage as a traditional C++ pointer. The ObjectStar class defines five types of member functions: constructors, destructors, dereferencing operators, assignment operators, and equivalence operators.

Constructors and Destructors ObjectClass defines three constructors and one destructor:

```
ObjectStar();
ObjectStar( Object * );
ObjectStar( const ObjectStar & const );
~ObjectStar();
```

The first constructor initializes `_pointer` to zero; it allows programs to define arrays of ObjectStars, since the classes of elements of arrays in C++ must provide a parameterless constructor[13]. The second and third constructors store their parameter value in `_pointer` and call `_pointer->reference()` if `_pointer` is non-zero. The destructor calls `_pointer->unreference()` if `_pointer` is non-zero.

Assignment Operators ObjectClass defines two assignment operators:

```
ObjectStar& const operator=( Object * );
ObjectStar& const operator=( const ObjectStar & const );
```

Both perform the operations of the destructor and then the operations of the constructor that takes the same type of parameter.

Dereferencing Operators ObjectClass defines two dereferencing operators:

```
operator Object *() const;
Object * operator->() const;
```

The `Object *` operator, which converts an ObjectStar to a traditional pointer, is intended to be used only by assignment operators. Operator -> supports the invocation of the member functions of the object pointed to by `_pointer`.

Equivalence Operators ObjectClass defines two equivalence operators:

```
friend int operator==( const ObjectStar & const, const ObjectStar & const );
friend int operator!=( const ObjectStar & const, const ObjectStar & const );
```

Operator == returns true if and only if the values of the _pointer members of the two parameters are equal. Operator != returns true if and only if the values of the _pointer members of the two parameters are not equal.

3.4 Pointer Meta-hierarchy

To preserve the compile-time type checking features of C++, we developed a tool to build a meta-hierarchy of pointer classes that mirrors the hierarchy rooted by class Object. Each class in the meta-hierarchy defines constructors, assignment operators, and dereference operators analogous to those defined by ObjectStar. Neither a destructor nor equivalence operators need to be defined, since they can be inherited directly from class ObjectStar. Each subclass of ObjectStar, however, defines two additional constructors and two additional assignment operators:

```
SubclassStar( Object * );
SubclassStar( const ObjectStar & const );
SubclassStar& const operator=( Object * );
SubclassStar& const operator=( const ObjectStar & const );
```

Besides performing the operations of the similar functions described in Section 3.3, these constructors and assignment operators perform a downward pointer cast[1] and a run-time check (see Section 4) of the safety of the cast.

4 Classes as Instances

The motivation for representing classes as run-time instances in *Choices* arises from several requirements.

- The object-oriented application interface supports the dynamic creation and removal of objects in the kernel and user space. References to the objects may be passed from object to object. Class enquiry functions allow simplified programming, monitoring, and class-based, run-time controllable debugging.
- Instances of new subclasses of system abstract classes provide a mechanism to extend the application interface in a controlled but non-trivial manner. Classes provide query functions that return their superclass and subclasses, documenting the state and evolution of the system and providing a database of system facilities.
- Instances of a class can be listed providing a database of system services that conform to the protocol of the class.
- The classes of objects are represented at run-time as objects. This allows run-time checking[2] that the protocols used in messages to objects are consistent with their definition.
- The classes of persistent objects can be recorded as persistent objects.

[1] There are few cases in which a downward pointer cast is appropriate, one such case we have found is the retrieval of persistent objects from name servers and object stores.

[2] C++ provides safe and efficient compile-time type checking; there are certain rare case, however, when it is too restrictive. Also, one can use run-time type checking to double-check compile-time type checking.

4.1 Implementation

First-class classes or class objects are implemented in *Choices* using a class called *Class*. Classes are similar to the *Dossiers* described in [6], except that Classes also support dynamic code linking and portable debugging. At run-time, every significant object in the *Choices* system is associated with a specific Class. The following code defines the interface for Classes:

```
typedef ObjectStar (*CP)( ObjectStar & const );

class Class : public Object {
                ...
public:
        Class( ClassStar & parent, const char * name, CP constructor = 0 );

        virtual setConstructor( CP constructor );
        virtual ObjectStar constructor( ObjectStar & );

        virtual ClassStar classOf();
        virtual int isA( ClassStar );

        virtual const char * name();

        virtual ClassStar parent();
        virtual ClassStar child();
        virtual ClassStar sibling();

        virtual int displayMembers( OutputStreamStar );
        virtual int displayKindred( OutputStreamStar );

        virtual void setInitDebugMask( char mask );
        virtual void setInitDebugMessages( short messages );
        virtual void setInitDebugObjects( int objects );

        virtual int debugMembers( char mask, short messages );
        virtual int debugKindred( char mask, short messages );
}
```

The typedef defines pointers to functions we call *addressable constructors* that initialize dynamically loaded class instances (see Section 5). The Class constructor takes a pointer to the Class's superclass, which is zero for the root of a hierarchy, and a character string name of the class. The constructor has an optional third parameter, which is a pointer to an addressable constructor. If a class does not have an addressable constructor or if its definition is deferred until run-time, then the third argument can be omitted. If an addressable constructor is supplied at run-time, a CodeLoader (see Section 5) may use the setConstructor operation to bind future invocations of the Class's constructor to invocations of a specific addressable constructor. The constructor operation provides a dynamically-bound constructor that will initialize an instance of the class corresponding to the Class.

The classOf operation returns a pointer to the Class of which the object is an instance. The isA operation returns true if and only if the parameter is a superclass of the Class.

The class hierarchy is modeled as a tree of arbitrary degree but it is implemented as a binary tree. Each Class object has a parent (superclass), a child (subclass), and a sibling (the next class with the same superclass). By convention, a zero indicates the parent of the root of a hierarchy, the child of a Class with

no subclasses, the sibling of a Class with no superclass, the sibling of a Class with a superclass that has no subclasses, or the sibling of a Class at the end of its superclass's sibling list.

Using Smalltalk terminology, an object is a *member* of the class from which it was instantiated. An object is said to be a *kind* of the class of which it is a member, and of all the class's superclasses. The displayMembers operation displays on the OutputStream the names of all objects that are members of this Class. The displayKindred operation displays on the OutputStream the names of all objects that are kind of this Class.

The next five methods help support the portable debugging features described in Section 6. The setInitDebugMask, setInitDebugMessages, and setInitDebugObjects operations set the Class's initial Raid mask, message-count, and object-count to given values. The mask and message-count are used to initialize future objects that will be instantiated from this class. The object-count determines how many objects will be initialized with the current mask and message-count. If the object-count is zero, object's belonging to this Class will be given zero masks and message-counts, regardless of their current values.

The debugMembers operation sets the Raid mask and Raid message-count of all member objects of the Class. The debugKindred operation sets the Raid mask and Raid message-count of all objects that are kind of this Class.

4.2 Usage

An auxiliary program, **allClasses**, coordinates the definition and instantiation of members of Class class; it is run automatically by the *Choices* "Makefiles." To simplify the allClasses program, we adopt a programming convention that requires certain lines of code to be present in both the header (".h") and source (".cc") files of a class. For example, consider the class *Object*. It is defined in "Object.h" and implemented in "Object.cc". The lines added to define a first-class class for Object in file "Object.h" are:

```
extern Class * ObjectClass;

class Object ... {
        ...
public:
        Class * classOf();
        ...
};
```

The lines define a pointer to the Class that is assigned a value at run-time. The value is used by the classOf implementation shown below. The lines needed in file "Object.cc" are:

```
Class *
Object::classOf()
{
        return( ObjectClass );
}
```

4.3 Displaying the Class Hierarchy from the System Shell

Each Class object is also bound to a name in the kernel Domain's NameServer. A user of the system shell can request information about any Class's place in the hierarchy. The Class is looked up by name in the NameServer, and the requested information is displayed.

There are three commands for displaying class hierarchy information:

1. **ancestors**, which recursively displays the superclasses of the given class;
2. **descendents**, which recursively displays the subclasses of the given class; and

3. **hierarchy**, which recursively displays first the superclasses and then the subclasses of the given class.

An example of the **hierarchy** command, which shows both the ancestors and descendents of Class *UNIX-Inode*, is shown below:

```
Choices> hierarchy UNIXInode
Object
  MemoryRange
    MemoryObject
      PersistentMemoryObject
        FileObject
          UNIXInode <<<
            BSDInode
            SVIDInode
```

4.4 Displaying Instances of Classes from the System Shell

All objects that are instances of subclasses of class *Object* are inserted into a global object when constructed and removed from the table when deleted. Class objects have access to this table and can iterate through the table searching for instances of the class that they represent.

The system shell uses the Smalltalk terminology both for displaying instances of a class and for Raid debugging. There are two commands for displaying instances of classes:

1. **members**, which displays all instances of the given class, and
2. **kindred**, which displays all instances of the given class and its descendants.

Examples of these commands follow:

```
Choices> members MemoryObjectPartition
MemoryObjectPartition[458100](:0:2)
MemoryObjectPartition[458040](:0:0)
2 instances.

Choices> kindred UNIXInode
BSDInode[466000](/)(:0:0:2)
BSDInode[466100](lost+found)(:0:0:3)
SVIDInode[45a180](/)(:0:2:2)
3 instances.
```

5 Adding Subclasses and Instances to a Running C++ Program

Choices provides an extensible application interface that permits reconfiguration and the addition of new services and facilities to the kernel without rebooting the system. Implementing an extensible kernel as a dynamic collection of objects allows a minimal kernel to be defined for *Choices*. New features may be added to the kernel to customise it for applications or for the target environment. Existing software in the system may invoke methods on these new services if they are subclassed from an abstract class known to the system at compile time. New software added to the system may use all the new methods of the new services. C++ has neither dynamic linking of code nor dynamic binding for constructors. To simplify and generalize the mechanism involved, the base classes of the system were extended to support the dynamic loading and linking of new subclasses and their methods using a scheme that has been used by several other research groups[3].

To access newly loaded code from an existing, running program, the running program needs a dynamic binding mechanism that maps messages (function calls) to invocations of the new code. C++ has an ideal mechanism for performing this dynamic binding: the virtual function table which is accessible through an object's *vtable* pointer. Once an object has been constructed, the member functions can be invoked through the object's vtable pointer. However, C++ constructors, which assign vtable pointers to objects at run-time, are ***statically bound*** and may not be called directly from an existing, running program. The problem to be solved in an implementation of dynamic, loadable C++ code is to allow existing programs to call the constructors of the newly loaded objects.

5.1 Dynamically-bound Constructors

Each class in the *Choices* system is represented by a *Class* object, described in Section 4, which is accessible through NameServers. Our solution to invoking traditional static C++ constructors dynamically uses a constructor method, which indirectly invokes the traditional static C++ constructor and is defined for *Class* objects.

We distinguish three interrelated constructor functions: traditional C++ constructors, which have names like Object::Object, addressable constructors, which we name in the manner ObjectConstructor, and a dynamic constructor, which we name Class::constructor. Traditional constructors are no longer called directly by code that is designed to take advantage of the dynamic linking facility. However, they are still used to allocate and initialize heap storage.

An addressable constructor, which is implemented as a C++ function, is linked and loaded at the same time as the new class to which it will be bound. The linker can thus bind an invocation of the traditional constructor into the code for an addressable constructor. Unlike traditional constructors, addressable constructors return a zero in case of failure. A failure may occur if the class of the parameter to the addressable constructor was inappropriate (because a run-time check failed.)

A dynamic constructor is implemented by the constructor member function of class Class. It provides dynamic linking to the traditional static constructor function by an invocation of the addressable constructor to which the static constructor has been linked. When the code for the new class is to be loaded, an instance of class Class is created to represent the new class. When the code is loaded, the linker installs the address of the addressable constructor in the Class object's _constructor instance variable. When the dynamic constructor of the Class object is invoked, it, in turn, invokes the addressable constructor stored in the _constructor instance variable. The function then invokes the static constructor.

In detail, the following steps are performed by the dynamic constructor:

1. If the _constructor variable is zero, it creates an instance of class CodeLoader (described below) and initializes it with the name of the Class.
2. Since the CodeLoader object performs all its work when created, it then deletes the CodeLoader object.
3. If the _constructor variable is still zero, then it fails by returning zero.
4. If the _constructor variable is non-zero, it calls the addressable constructor, which calls the traditional constructor. The return value of the traditional constructor is also returned by the addressable constructor, which is then returned by the dynamic constructor.

5.2 The CodeLoader

Choices defines a subclass of the abstract class CodeLoader for each type of object file format used by the project. The first such class that we implemented supported the Common Object File Format[4]; thus the subclass is named COFFLoader. The CodeLoader constructor performs the following operations:

1. open a file that contains the symbol table for the running program.

2. look up the name of the class with the undefined constructor in a NameServer that maps class names to file names.

3. if the file name is found, open the corresponding file, which should be a relocatable, unstripped module that includes the code for the class and for closely related classes.

4. allocate memory and read code from file.

5. resolve undefined symbols in the loaded code with the symbols found in the running program's symbol table.

6. relocate symbols in the loaded code, basing them on the memory address at which the code is loaded.

7. create Class objects for any loaded classes that do not already have them.

8. install addresses of addressable constructors defined in the loaded code into corresponding Class objects.

5.3 Evaluation

By making loadable C++ objects available in the support for *Choices*, parts of the operating system can be greatly simplified. For example, the *Choices* file system supports UNIX 4.3 BSD, System V, and MS-DOS files. File system components that are needed can be loaded when a request is made to access a file. Thus, neither the kernel nor user space is penalised for the flexibility that is provided by the file system. A further benefit of this approach in the file system is that it makes loading the class methods for persistent objects that are activated simple to implement. Further, user-defined persistent objects may be added at any time to the collection of persistent objects in the system using the same mechanism. The C++ language could have provided improved support for loadable objects if invocations of constructors could be linked at run-time.

6 Portable Debugging

Debugging the kernel of an operating system is not a pleasant task, even if it is object-oriented. The problem is compounded by a lack of a kernel debugger that can report errors in terms of the objects and classes of which the system is composed. Debugging systems for C++ do exist, but most of them are proprietary, making them unsuitable for use in our activities. Further, our system is implemented on a wide variety of architectures. Because of compiler availability, for some systems we use the Gnu C++ compiler, and in other cases we use AT&T's Cfront. The Gnu symbolic debugger gdb allows symbolic debugging of programs, but this system only gives rudimentary support to the debugging of classes and instances. The lack of debugging facilities and non-uniformity of the debugging environment encouraged us to build debugging into the system as a feature that we call Raid, which is the brand name of some household pesticides.

Raid implements portable, object-oriented debugging in *Choices*. It is implemented as a set of methods, instances, and statements that are included in the *Choices* source code and operate by printing the values of expressions at run-time on the *Choices* console. Because the debugging facilities exist in objects at run-time, it provides a programmable debugging interface. Various debugging features can be turned on or off dynamically at run-time using *Choices* applications, like the *Choices* command interpreter or standard debugging tools like gdb. Further, a library of debugging utilities has been built up to provide more sophisticated debugging tools. Raid provides a much more flexible debugging environment than one based on placing diagnostic print statements in the code because it is integrated with the *Choices* object-oriented architecture.

6.1 Raid Masks

Raid supports five different categories of debugging statements for either existing instances or future instances of specific classes or classes and their descendents. These debugging categories are shown in Figure 1 along

Category	Raid Mask Name	Bit	Mask
constructor/destructor functions	`RaidConstructor`	0	1
reference()/unreference()/noRemainingReferences() functions	`RaidReference`	1	2
public member functions	`RaidMethod`	2	4
protected and private member functions	`RaidProtected`	3	8
detailed information within any type of member function	`RaidDetail`	4	16
all of the above	`RaidAll`	0–4	31

Figure 1: Raid Debugging Masks

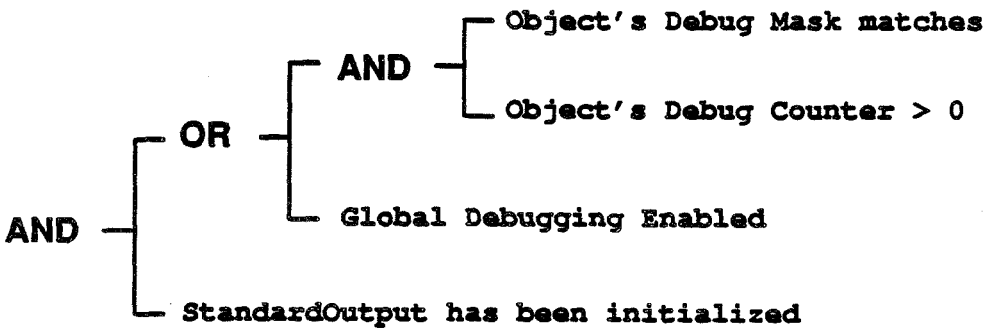


Figure 2: Raid Logic

with the six bit masks that are used to encode them in Raid. The categories reflect common debugging problems that have arisen in *Choices*.

Masks can be combined (using a logical sum) in *interactive debugging commands*. For example, the *Choices* system shell (command interpreter) includes the odb or cdb statements (see below) that, with the mask "`RaidConstructor|RaidReference`", can be used to view constructor, destructor, and reference counting method invocations. For convenience, masks can either be specified as bit masks or as names (optionally abbreviated) enclosed in double quotation marks. Code that is instrumented with Raid statements use a single named mask to indicate under what conditions the information from the instrumentation is to be collected. The name used is given in Figure 1.

6.2 Raid Implementation

C++ offers both the well-known C-style printf statement that some C++ developers prefer or the new C++-style << operator. Raid supports both means of printing debugging information:

- Raid(mask)() supports printf-style debugging, and
- Raids(mask) supports <<-style debugging.

Raid has a simple implementation — a pair of macro definitions, which use the logic given in Figure 2.

6.3 Instrumentation

Raid does not automate the inclusion of debugging statements, the developer must still insert these statements at useful places within the code. From experience we have adopted the convention of inserting Raid statements

at three places within member functions: on function entry (to display the arguments with which the function was called), before each return statement (to display the return value(s) of the function), and between or within blocks of code (to display detailed information about the current function invocation).

Because Raid is part of the *Choices* system, one must exercise some restraint to avoid writing expressions in debugging statements that might have *Choices* side-effects. For example, the "++" or "--" operators will cause side-effects if used within Raid statements.

Constructor Instrumentation The beginning of each constructor should call the initDebug method. This initializes the object's mask and message count with the values set in its Class. Constructors may also include Raid statements on entry and exit, like the examples given below for member functions.

Function Entry Instrumentation Function entry can be instrumented by inserting a Raid statement at the beginning of the function. This Raid statement should be labeled with the appropriate mask. For example, one can instrument the beginning of the following member function,

```
ReadFileStream::read( char * buffer, int length )
```

using the Raid mask "RaidMethod" with either of the following Raid statements:

```
Raid( RaidMethod )( "%N->ReadFileStream::read( %x, %d )\n", this, buffer, length );
            or
Raids( RaidMethod ) << this << "->ReadFileStream::read( "
                    << asHex( buffer ) << ", " << length << " )\n";
```

The various Raid masks allow functions to be classified into the categories given in Figure 1.

Function Exit Instrumentation In a similar manner, function exit can be instrumented by inserting a Raid statement before the return of the function. For example, before the return statement,

```
return( bytesRead );
```

the code can be instrumented by a Raid statement with either of the following forms:

```
Raid( RaidMethod )( "%N->ReadFileStream::read returns %d\n", this, bytesRead );
            or
Raids( RaidMethod ) << this << "->ReadFileStream::read returns " << bytesRead << "\n";
```

The mask for entry and exit should match.

Detailed Debugging Complex code requires a more detailed debugging instrumentation that provides information on the status of variables within a function. A developer may add other Raid statements throughout a function body. The "RaidDetail" mask is used for this purpose.

6.4 Controlling Raid output from the System Shell

Raid output can be controlled from both the *Choices* shell and from kernel or application code. This section describes the control of Raid from the system shell.

There are two commands for controlling Raid debugging:

- odb, which turns on debugging for existing objects belonging to a given class, and
- cdb, which turns on debugging for future objects belonging to a given class.

The commands allow debugging information to be selected on a class basis for particular categories of instrumentation:

```
Choices> odb {member|kind} ClassName mask [messageCount]
```

and:

```
Choices> cdb {member|kind} ClassName mask [messageCount [objectCount]]
```

For both commands, a mask of zero will turn off debugging. For example, the following command turns on debugging for the public methods of all the currently instantiated members of *ReadFileStream*:

```
Choices> odb member ReadFileStream "RaidMethod"
```

and the next command turns on all the debugging of all the currently instantiated members of *FileStream* and its subclasses and lists the first 10 messages from each object:

```
Choices> odb kind FileStream "RaidAll" 10
```

Experiences with debugging *Choices* led us to add a Raid feature for examining particular activities. The cdb command allows instrumentation to be turned on for objects that are created as a result of some newly started activity.

For example, the following command turns on constructor and reference count debugging for all future objects instantiated from class *WriteFileStream*:

```
Choices> cdb member WriteFileStream "RaidConstructor|RaidReference"
```

The count parameters may be used to turn on all debugging for 10,000 messages for each of the next, say, 3 objects instantiated. For example, the following command turns on debugging from class *FileStream* and its subclasses:

```
Choices> cdb kind FileStream "RaidAll" 10000 3
```

6.5 Controlling Raid from Kernel or Application code

Debugging instrumentation can be controlled from C++ code by three operations on *Objects*:

1. setDebugMask(char mask)
2. setDebugMessages(short messages)
3. setName(const char * str)

The setDebugMask operation sets the object's Raid mask to the given value. A mask of zero will turn off debugging for the receiving object. The setDebugMessages operation sets the object's message-count to the given value. A message-count of zero will turn off debugging for the receiving object. An object must have a non-zero mask and message-count before it will print debugging messages. The setName operation sets the object's "nickname" to a copy of the given character string. The nickname is printed out with each Raid statement.

In *Choices* objects may be debugged using Raid in either system or user space. This is facilitated by the *Choices* ObjectProxy mechanism that makes invocations of method calls to kernel and user objects transparent to the calling program.

6.6 Using Raid with a Symbolic Debugger

Although Raid is useful by itself, it becomes a more powerful and flexible tool when used with a symbolic debugger. Advanced symbolic debuggers allow debugger commands to be executed at breakpoints and permit messages to be sent from the user to the objects of the program at run-time. The implementation of the symbolic debugger we use for system programming is independent of the *Choices* system console and so it can be used to debug the early stages of a *Choices* port to a new architecture. More importantly, the symbolic debugger allows the user to interact with Raid at run-time, based on the execution of specific breakpoints. Debugging information output can be selected by setting masks and counters interactively. This facility allows the behavior of a class hierarchy or instance to be examined at a specific point in the program. In boot code, it also allows Raid to be used interactively without input from the console.

For instance, using a symbolic debugger, one can define a macro that enables debugging in a region of a source file delimited by a starting and an ending line. The macro sets a breakpoint at the beginning of the region and another one at the end of the region. It also associates debugger commands with each breakpoint that turn on or off the global Raid debugging switch and continue execution. The whole process is invisible to the user who sees only the output of the Raid statements inside the region. This approach allows control of debugging on a method or even individual Raid statement basis.

7 Summary

During the construction of the *Choices* operating system, we discovered that several facilities can be of considerable importance and can lead to great code simplification in developing an object-oriented operating system. The facilities are not supported in C++, the *Choices* implementation language. However, by careful organization, we could introduce the facilities by building a suitable set of base classes and tools.

The facilities we found of value are:

- the automatic deletion of unreferenced objects,
- the representation of a class by an object at run-time,
- the dynamic loading of new subclasses and code, and
- support for debugging classes and instances at run-time.

The effect of these facilities was to speed up debugging and simplify the code of many parts of the operating system; for example, the file system, application interface, name-servers, and persistent object code.

In conclusion, we would like to draw the attention of language designers to the provision of these facilities in an object-oriented systems programming language. It remains an open question whether these facilities are best provided directly by the language or the language should support the construction of these facilities. Our experience with C++ shows that some of the mechanisms needed to implement the facilities could not be programmed directly in the language. Instead, we had to build tools to augment the C++ language.

References

[1] Roy H. Campbell and Peter W. Madany. Considerations of Persistence and Security in Choices, an Object-Oriented Operating System. In *International Workshop on Computer Architectures to Support Security and Persistence of Information*, pages 12.1–12.14, University of Bremen, Federal Republic of Germany, May 1990. The proceedings also to appear in the Springer-Verlag Lecture Notes Series.

[2] Roy H. Campbell, Vincent Russo, and Gary Johnston. Choices: The Design of a Multiprocessor Operating System. In *Proceedings of the USENIX C++ Workshop*, pages 109–123, Santa Fe, New Mexico, November 1987.

[3] Sean M. Dorward, Ravi Sethi, and Jonathan E. Shopiro. Adding New Code to a Running C++ Program. In *Proceedings of the USENIX C++ Conference*, pages 279–292, San Francisco, California, April 1990.

[4] Gintaras R. Gircys. *Understanding and Using COFF.* O'Reilly and Associates, Inc., Newton, Massachusetts, 1988.

[5] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

[6] John A. Interrante and Mark A. Linton. Run-time Access to Type Information in C++. In *Proceedings of the USENIX C++ Conference*, pages 233–240, San Francisco, California, April 1990.

[7] Gary M. Johnston and Roy H. Campbell. An Object-Oriented Implementation of Distributed Virtual Memory. In *Proceedings of the Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pages 39–57, Ft. Lauderdale, Florida, October 1989.

[8] Peter W. Madany, Roy H. Campbell, Vincent F. Russo, and Douglas E. Leyens. A Class Hierarchy for Building Stream-Oriented File Systems. In Stephen Cook, editor, *Proceedings of the 1989 European Conference on Object-Oriented Programming*, pages 311–328, Nottingham, UK, July 1989. Cambridge University Press.

[9] Vincent Russo and Roy H. Campbell. Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems using Class Hierarchical Design. In *Proceedings of OOPSLA '89*, pages 267–278, New Orleans, Louisiana, September 1989.

[10] Vincent F. Russo. *An Object-Oriented Operating System.* PhD thesis, University of Illinois at Urbana-Champaign, October 1990.

[11] Vincent F. Russo, Peter W. Madany, and Roy H. Campbell. C++ and Operating Systems Performance: A Case Study. In *Proceedings of the USENIX C++ Conference*, pages 103–114, San Francisco, California, April 1990.

[12] Robert Seliger. Extended C++. In *Proceedings of the USENIX C++ Conference*, pages 241–264, San Francisco, California, April 1990.

[13] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, 1986.

To be presented at the Twenty-fifth Hawaii International Conference on System Sciences, Kauai, Hawaii, January 7-10, 1992.

NASA
Sy-61
93867
D45138
26 p.

Organizing and Typing Persistent Objects within an Object-Oriented Framework*

Peter W. Madany and Roy H. Campbell
University of Illinois at Urbana-Champaign
Department of Computer Science
1304 W. Springfield Avenue, Urbana, IL 61801 USA

Email:	madany@cs.uiuc.edu	roy@cs.uiuc.edu
Phone:	1-217-333-7937	1-217-333-0215
Fax:	1-217-333-3501	1-217-333-3501

Abstract

Conventional operating systems provide little or no direct support for the services required for an efficient persistent object system implementation. We have built a persistent object scheme using a customisation and extension of an object-oriented operating system called *Choices*. *Choices* includes a framework for the storage of persistent data that is suited to the construction of both conventional file systems and persistent object systems. In this paper we describe three areas in which persistent object support differs from file system support: storage organization, storage management, and typing.

Persistent object systems must support various sizes of objects efficiently. *Customizable containers,* which are themselves persistent objects and can be nested, support a wide range of object sizes in *Choices*. Collections of persistent objects that are accessed as an aggregate and collections of light-weight persistent objects can be clustered in containers that are nested within containers for larger objects. Automated garbage collection schemes are added to storage management and have a major impact on persistent object applications. The *Choices* persistent

*This work was supported in part by NSF grant CISE-1-5-30035 and by NASA grant NAG 1-163.

object store provides extensible sets of persistent object types. The store contains not only the data for persistent objects but also the names of the classes to which they belong and the code for the operations of the classes.

Besides presenting persistent object storage organization, storage management, and typing, this paper discusses how persistent objects are named and used within the *Choices* persistent data/file system framework.

1 Introduction

Efficient persistent object system implementations require services that are not usually found in conventional operating systems. We have built a persistent object scheme that is supported by the specialized services of a customizable and extensible object-oriented operating system called *Choices*. *Choices* includes a framework for the storage of persistent data[9] that is suited to the construction of both conventional file systems and persistent object systems. In a previous paper[5] we discussed *Choices* and how it provides security for persistent objects. In this paper we concentrate on three other areas in which persistent object support differs from file system support: storage organization, storage management, and typing.

Persistent objects incorporate the operating system notion of persistent data storage and the programming language notion of objects[4]. Many conventional operating systems store persistent data in files[2]. Persistent objects in *Choices* differ from regular data files in three fundamental ways:

- they provide a persistent data abstraction, encapsulate data, allow user defined operations, and impose a notion of type. Usually, the operations defined for regular data files are only read and write.

- their operations can store, retrieve, and dereference references to other persistent objects.
- invocation of an operation ensures that the encapsulated persistent data is retrieved. Both the retrieval and storage of data are automated and do not require explicit opens, reads, writes, and closes.

Providing the notion of a persistent object as an operating system service allows it to be used as a powerful concept with which to build other, both conventional and unconventional, services. The three features of a persistent object allow, for example, the convenient programming of a conventional file directory as a persistent object in *Choices*. Operations allow the contents of a directory to be listed in a readable format, to be searched for the location of a particular filename, or to be updated with the addition or removal of a filename.

Choices is not limited to a single application programming language. However, currently we have implemented persistence in *Choices* only for objects that are programmed using C++[16]. In addition, these persistent objects have the following restrictions:

- they must be instances of subclasses of the PersistentObject class, and
- they can store references only to other persistent objects.

Besides presenting persistent object storage organization, storage management, and typing, this paper discusses how persistent objects are named and used within the *Choices* persistent data/file system framework.

2 Persistent Data Storage Framework

The *Choices* file system framework contains a hierarchy of classes that can be combined to build both standard and customized storage systems. The framework is flexible enough to support both persistent storage systems and traditional file systems efficiently[14].

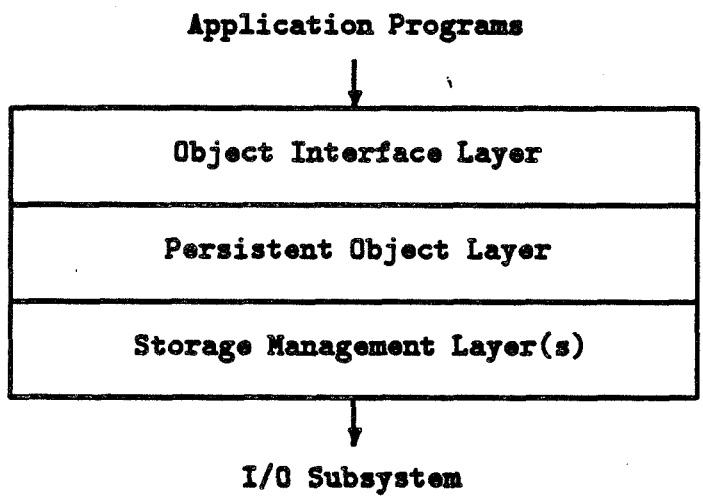


Figure 1: Layers of the File System Framework.

The framework divides a persistent data storage system into a minimum of three layers as shown in Figure 1. Each layer of the framework contains or uses objects that belong to one of two fundamental classes:

- the PersistentStore class defines persistent data stores, each of which store and retrieve persistent data using a random access method, and
- the PersistentObject class defines persistent objects, which encapsulate and provide operations on the data managed by a persistent store.

The PersistentStore provides random access to an uninterpreted sequence of data while the PersistentObject interprets the data of a persistent store as having a format.

Objects in the layers of the file system serve the following purposes:

- Objects in the *Storage Management Layers* access hardware devices like disk controllers or network interfaces in the I/O subsystem. Access is defined in terms of sequences of units, where a unit corresponds to the physical block size used by the hardware device. Objects in the higher *Storage Management Layers* manage the contents of the underlying physical persistent stores, interpreting the data in the stores as nested containers of logical stores. The Storage Management Layers contain PersistentStores and PersistentObjects that encapsulate storage organization and sharing.

- Objects in the *Persistent Object Layer* encapsulate the data within persistent stores and provide operations on the data. These operations include support for naming and data structuring. System programmers can extend the framework by designing persistent objects that support many types of operations and inter-object relationships. All objects in the Persistent Object Layer are PersistentObjects. These objects may be accessed by applications directly; however, application programs can also access them through objects in the Object Interface Layer.

- Objects in the *Object Interface Layer* define additional access protocols for both application and system programs. They use naming objects in the Persistent Object Layer to provide a unified name space for referencing objects in the system. They also provide higher level access protocols for the data encapsulated within the various types of files in the Persistent Object Layer. All objects in the Object Interface Layer refer to PersistentObjects or PersistentStores.

The PersistentStore class defines an access protocol for both physical storage devices and logical storage devices; each storage device is modeled as a sequence of identically sized blocks of persistent data.

The most important data access operations defined by PersistentStores are read and write. These operations retrieve or store one or more contiguous blocks of data, and they are used as primitives by other access methods. For example, clients of the file system may use the abstract protocols of a PersistentObject to access the persistent data of a PersistentStore.

Each PersistentStore has an associated PersistentObject class that provides a data abstraction and encapsulation of the persistent data in the store. At run-time, there is a one-to-one correspondence between an instance of a PersistentStore and its associated PersistentObject. The PersistentStore asA method returns a reference to the store's PersistentObject. If the PersistentObject has not yet been instantiated, the method creates the object by memory mapping the store's persistent data and using the data as part of the encapsulated data of the object. The PersistentStore thus provides the underlying data for its associated PersistentObject. PersistentObjects and their underlying PersistentStores provide the framework and foundation for the *Choices* file system and persistent object store.

In more detail, PersistentObjects and their underlying PersistentStores implement *object-oriented* access to persistent data. The asA method takes an argument which may be either a concrete or an abstract Class (see Section 5) and returns a reference either to an instance of the argument or an instance of a concrete subclass of the argument, respectively. The PersistentStore supports method checks to ensure that the store's associated PersistentObject class (and underlying data representation) is compatible with the class of the object whose reference is returned.

Several file system or persistent object store clients may access the same persistent data. To pro-

vide data consistency for concurrent updates to persistent data through the methods of a persistent object, the PersistentStore ensures that there is, at maximum, only one instance of its associated PersistentObject. Several PersistentObject subclasses use this data consistency provision to implement a main memory cache for frequently accessed persistent data. When a PersistentObject is no longer needed in primary memory, its finalization code calls the close operation on its underlying PersistentStore. A further asA request will create a new PersistentObject that uses the existing persistent data.

The concept of a PersistentStore is used both for physical and logical storage devices allowing reuse of code. All concrete PersistentStores (see Figure 3) belong to one of the following two subclasses:

- Disks encapsulate physical storage devices like hard disk drives, floppy disk drives, and RAM disks. Disks communicate with objects in the I/O subsystem.
- Files encapsulate logical storage devices like UNIX inodes and disk partitions. Files communicate with objects in a lower Storage Management Layer of the file system.

Several Disk subclasses of PersistentStore contain machine-specific code to interface with disks and controller hardware. In "conventional" operating systems, the methods of these classes would be the disk driver routines. The associated PersistentObject classes are used to structure the data on the disks into collections of partitions and collections of files. One example of such a PersistentObject is the PersistentStoreContainer that is used to contain files.

A File is a PersistentStore that is contained within a PersistentStoreContainer. Instead of containing hardware interface code like the disks, it has a source PersistentStore that supplies it with data from a lower layer of the file system. Files provide a *window* into their source PersistentStore.

The size of this window can be either fixed or variable and can range from zero up to the size of the source PersistentStore. The window can be either contiguous or divided into various discontiguous regions of blocks.

Recursively, the data managed by a File can have structure and can be organized into a collection of files. Again, in such cases a subclass of PersistentStoreContainer will be associated with the PersistentStore. In general, a PersistentStoreContainer manages the contents of a PersistentStore as an indexed collection of contained Files. Its methods create, manage, and delete contained Files.

PersistentStoreContainers can be nested to an arbitrary depth. The nesting supports the multiple Storage Management Layers of the framework and also permits user-defined file systems, archives and persistent stores. The PersistentStoreContainer associated with the lowest, physical device, layer divides a disk into a collection of partitions. Each partition is a PersistentStore in the next "logical file system" layer. A PersistentStoreContainer is associated with the PersistentStore in this layer and structures the data of a partition into a collection of files according to the specification of a particular file system. For example, a specific subclass of PersistentStoreContainer structures the data in a partition into a collection of inodes using the inode system format of System V. At higher levels, a file can have an associated PersistentStoreContainer that imposes a *tar* format on the data in a file and provides the abstraction that the file is a collection of tape archived files.

In this section we have described the two main components of the *Choices* file system and persistent object store; the PersistentStore and PersistentObject. Each PersistentStore has an associated PersistentObject which encapsulates the persistent data in the store and provides an abstraction of that data. Access to the PersistentObject conforms to an object-oriented approach using subclasses and inheritance. The PersistentStoreContainer is a particularly important PersistentObject abstraction that structures persistent data into a collection of independent PersistentStores.

3 Storage Organization

Many file systems are optimized to store many small files and a few large files. For example, in UNIX, the implementation uses disk blocks and indirect disk blocks. However, file systems that are organized to use one or more disk blocks for a file can only provide "heavy-weight" support for persistent objects and perform better when the objects are one or more disk blocks in size. Such file systems allow concurrent access to a small maximum number of files; some files may need to be closed to open other files. To program consistently with persistent objects, the size of the objects may range from a byte to the size of a large file. Persistent object systems must support smaller objects efficiently. The number of persistent objects active at any one time may be very large.

Choices supports a wide range of object sizes by using *nested, customizable* PersistentStoreContainers. A container divides a storage device into an indexed collection of PersistentStores, which hold the data for persistent objects. Containers are similar to the *segments* of the Comandos system[11] or the *file objects* of the EXODUS system[6], which both support the clustering of persistent objects for efficiency. Containers differ in that they fit within a general file system framework and they can be nested to an arbitrary depth. Light-weight persistent objects can be clustered in containers that are nested within containers for larger objects. The *Choices* class hierarchy (see Figure 2) defines containers that can be tailored to suit the objects they are intended to contain. A container may contain objects that have temporal locality; that is, if one of the objects is accessed it is likely that they all will be accessed. Alternatively, a container may contain objects of the same class. A large number of persistent objects can be active at any one time.

Table 1 shows how the overhead and minimum storage requirements vary for different persistent data store implementations. The overhead and requirements shown depend only upon the

Container Type	Object Overhead	Minimum Size of Stored Object	Directory Overhead
Class-Specific	0	size of object	4
General	10	1 block	4
MS-DOS	21	1 or 2 blocks	11
ar	44	size of object	16
System V	64	1 or 2 blocks	16
BSD 4.3	128	1 or 2 blocks	key length + 8
tar	412	1 block	100
AIX	512	0 blocks	key length + 8

Table 1: File System Overhead

requirements of the particular storage technique. In order to be disk format compatible with each storage technique, *Choices* imposes no additional storage requirements.

The least overhead is obtained for a container of persistent objects that are all of the same class. *Choices* memory maps the persistent data in each persistent object. Access is provided by the methods defined for the class. The collection stores the class of the objects stored in the container and hence there is zero extra storage overhead imposed per object. The size of the persistent store is exactly the size of the persistent object. The directory overhead to hold the logical name of the object is four bytes.

A container that holds objects of different classes, a general container, must store the size, location and class of each object. This imposes an overhead of ten extra bytes per object. Since each object may be of a different size, it is convenient to store the object in at least a block of a persistent store. The directory overhead is not increased.

The MS-DOS file system imposes the least overhead per file and directory of the various "stream-oriented" file systems we have implemented. The overhead records the protection and size of the file and the starting location of the file on disk. A non-empty file requires at least one cluster (one or two disk blocks) of persistent store where a block is the sector size of the disk. The eleven

bytes of directory information contain a two-part symbolic name. System V requires an overhead of 64 bytes per persistent store in which it stores "inode" information including protection and size information and pointers to the direct, single-indirect, double-indirect, and triple-indirect blocks that represent the file. It also uses at least one block to store a non-empty file. The directory information includes a fourteen character symbolic name and a two byte reference to the "inode" information. The AIX "inode" structure includes room for 384 bytes of data, therefore a minimal, non-empty file could use zero additional blocks of storage.

In this section, we have reviewed the requirements for a persistent store to support persistent objects and shown that most conventional file systems are inappropriate and impose too much overhead and too many limitations. However, existing file system implementation techniques can be used to provide storage for persistent objects. In *Choices*, we have exploited file system implementations by building an object-oriented framework for persistent storage that can be subclassed both to support efficient conventional file systems and efficient persistent objects.

4 Storage Management

The storage management issues for files systems and persistent object stores differ significantly because persistent objects can store references to other persistent objects. In practice, our *Choices* implementation required different garbage collection algorithms for persistent object containers and file containers. Nevertheless, the implementation allows all the different kinds of containers to be incorporated within the same storage organization framework. Each container is customized to support specific file system or persistent object storage management behavior.

Many file systems restrict inter-object references to directories, and they further restrict di-

rectories so as to disallow cyclic references. This avoidance of cyclic persistent data structures allows traditional file containers to use reference-counting or even simpler storage management techniques. An example of the use of reference-counting for storage management is a container of UNIX inodes[17]. UNIXContainers allow a regular file to have many directory entries but restrict a directory to have only one directory entry. The inode for each file and directory has a *link count*, which holds the number of directories that reference it. Application programs can use the `unlink` system call to remove a file from a directory. This call decrements the inode's link count. When the link count reaches zero, the inode is deleted.

An example of the use of a simpler storage management technique is a container for MSDOS files[12]. MSDOSContainers allow a file or a directory to have only one directory entry. Therefore no reference count is needed. Application programs can use the `unlink` system call to remove a file from a directory. In the case of an MSDOSContainer, this call simply deletes the file.

Unlike the restricted cases of UNIX and MSDOS containers, general persistent object containers must allow arbitrary inter-object references. Therefore, more sophisticated garbage collection techniques are needed for containers of persistent objects. Persistent object containers can be considered as heap managers for the persistent objects they contain. Since we chose to use the C++[16] model for objects we were not limited by the language to a specific kind of garbage collection. Instead, we can select an appropriate garbage collection scheme depending on the use of a particular container. Currently we have experimented with two schemes: the first one uses reference counting and the second one does not.

Both garbage collection schemes are based on the concept of *reachable* objects. A container of PersistentObjects keeps track of which objects belong to a set of *root* objects. An object is defined to be reachable if it is a root object or if there is a chain of references to it from a root object.

All objects that are not reachable should have their storage reclaimed. The PersistentObject class defines two operations that enable applications to control which objects belong to the root set:

- persist, which adds the object to the set of root objects if it was not already in the set, and
- desist, which removes the object from the set of root objects if it was in the set.

Our first scheme, which was described in [5], supported applications that have references and persistent objects which can be modeled by a directed acyclic graph (DAG). This restriction is reasonable for many applications that avoid circular lists like our first applications. However, in general the objects and references of many applications cannot be modeled by a DAG and the first scheme is too restrictive.

The second scheme uses a separate process that periodically locates and removes unreachable objects. This process resembles the UNIX **fsck** program or the MS-DOS **chkdsk** program. It differs from these programs in three ways:

- unreachable objects are considered garbage as opposed to being considered "lost" files that have been "found,"
- the process must be run periodically instead of being used to recover from error conditions, and
- the process must have access to the layout of all classes of objects, not just the layout of directories.

In brief, our studies of persistent object stores and references indicate that garbage collection algorithms for persistent object stores are different from those for file systems. In *Choices*, every persistent object is accessed from a container and this container can be customized to support

specific storage management behavior. We have implemented three different garbage collection techniques, one for files, one for persistent objects that conform to a DAG reference model, and one that scans storage for garbage. The current *Choices* design for persistent objects accommodates the different storage management techniques within the same framework of classes. However, it remains to be seen whether improved storage management techniques can be embedded within the same framework. Furthermore, we have not implemented any scheme that would support garbage collection when inter-object references are allowed to span container boundaries, but we feel an approach like that proposed in [15] would be feasible.

5 Typing

Most conventional file systems have a small, fixed set of file types. However, persistent object stores must support extensible sets of persistent object types. The PersistentStore implementation allows extensible sets of persistent object types by associating persistent data with a Class of PersistentObject. *Choices* implements run-time type-checking to ensure that persistent data is accessed using the appropriate persistent object abstraction. The checking is performed by the supports method of the PersistentStore. The type checking is based on the C++ style of associating type with the notion of class. Inclusion polymorphism or class inheritance is supported by the type checking mechanism so that a client of the persistent object store may access an instance of a new subclass of persistent object using the protocols defined for an abstract, older superclass. Thus, the typing mechanism supports extensions and specializations of existing classes of persistent objects.

Invocation of a persistent object method automatically and dynamically loads the code of the methods for the class of the object. Again, older applications may execute methods of new classes

of persistent object using only the preexisting abstract superclass definitions. Thus by using an object-oriented approach, the type system supports the late binding and dynamic loading of code for persistent objects. These mechanisms are again captured as specializations of classes within a class hierarchy that is organized as a framework.

Classes from the *Choices* PersistentObject (see Figure 2) and PersistentStore (see Figure 3) class hierarchies can be combined to build several standard file systems such as System V or MS-DOS. Such file system standards account for many of the subclasses of PersistentStore. For efficiency, these PersistentStores are specialized to support the fixed set of associated PersistentObject classes required to implement the standard. For example, the BSDInode class can store the data for: regular files (PersistentArrays), two types of directories (BSDDirectories and HashedBSDDirectories), and symbolic links (SymbolicLinks). Because the set of associated persistent object classes is small and fixed, an efficient representation of the classes can be hard-coded into the BSDInode class. However, the class hierarchies also support system and user-defined persistent objects. In this case it is not possible to hard-code the set of PersistentObject classes associated with each PersistentStore. Therefore, the PersistentStore hierarchy includes the GeneralNode class and the PersistentObject hierarchy includes the GeneralContainer class.

The implementation language of *Choices*, C++, does not support the dynamic loading of new classes at run-time. However, as the provider of a persistent object store, *Choices* must support new persistent objects and new persistent data stores. Together the GeneralContainer and GeneralNode classes support the storage, retrieval, and use of new, user-defined persistent objects and data. The classes support the dynamic loading of other classes, avoiding the necessity to recompile the persistent object store.

The dynamic loading of new subclasses of PersistentObject and PersistentStore is accomplished

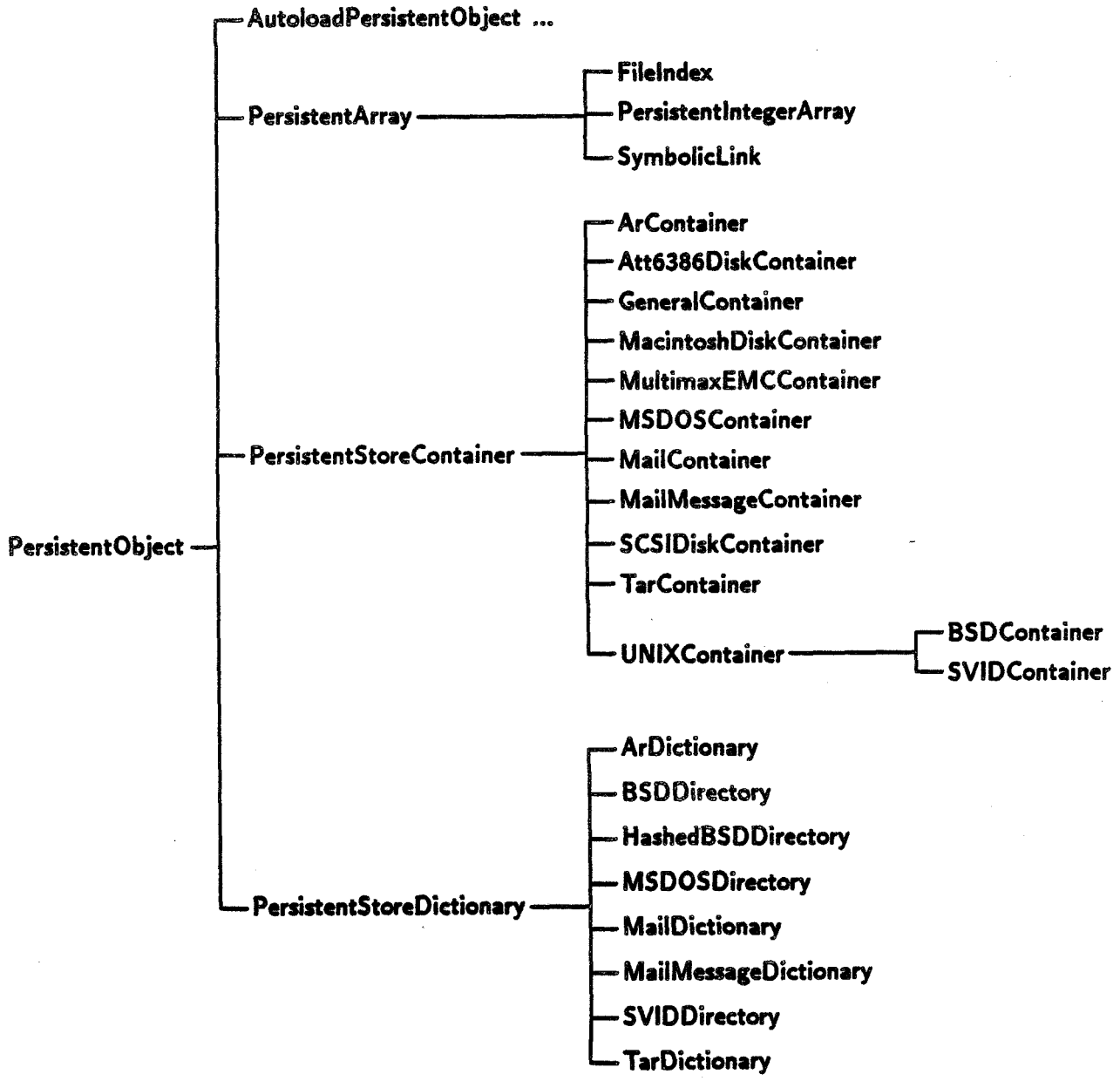


Figure 2: PersistentObject Class Hierarchy

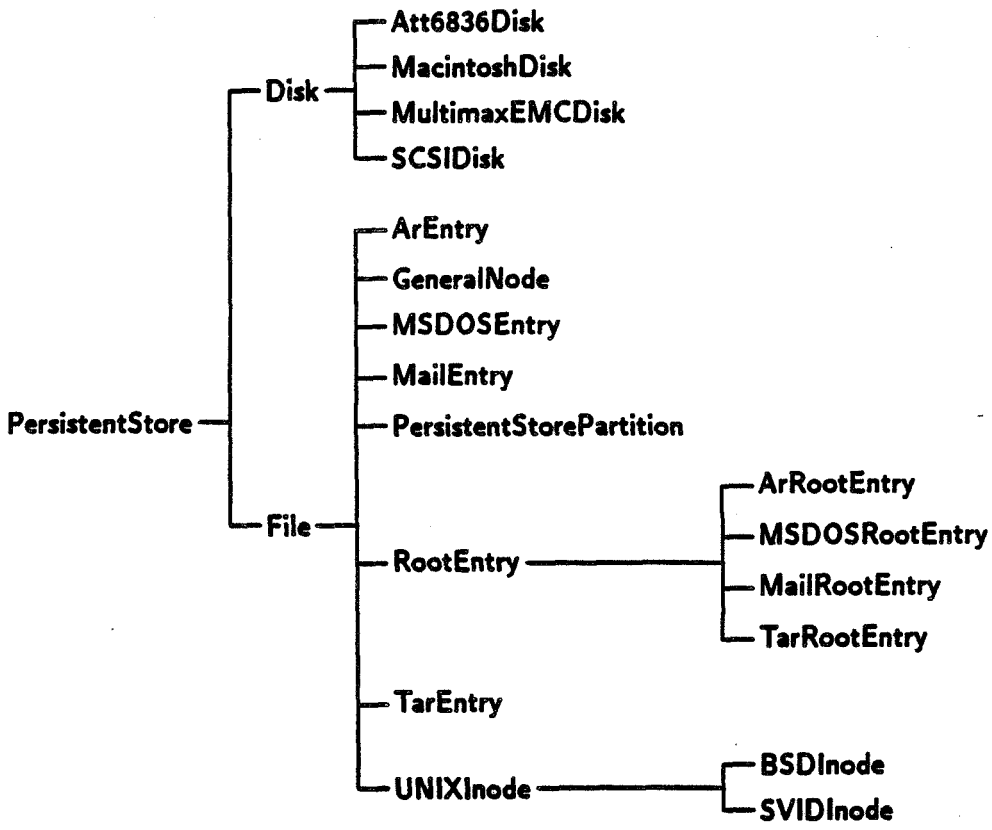


Figure 3: PersistentStore Class Hierarchy

using an object-oriented solution. In *Choices* we explicitly represent each class defined in the system as an object of class Class. Using object-oriented terminology, an instance of class Class is used to *reify* C++ class definitions. The GeneralContainer stores the name of the class of the PersistentObject that is associated with a GeneralNode as a character string. The GeneralNode refers to its associated PersistentObject class as an instance of class Class. The character string is converted to a reference to a Class by the lookup method of a kernel NameServer. The GeneralNode may invoke methods on the PersistentObject class using the protocol defined for class Class. This allows the GeneralContainer and GeneralNode classes to be compiled without using a specific class name for the associated PersistentObject, thus avoiding recompilation and relinking. For more information on first-class classes in C++, see [10, 8].

To support a dynamically extensible persistent object store, *Choices* must support the dynamic *loading* of code for new persistent objects. The new persistent object can be dynamically linked using the symbol table from previous loads and linkages. However, it is more difficult to modify the running system to invoke the constructor of the new object. Unfortunately, C++ constructors are *statically bound* at compile time and do not generate any useful symbolic linker and loader information. It is thus difficult to call the constructor from a running program like the persistent object store. Our solution to the dynamic invocation of the static C++ constructor involves indirection, using the invocation of an instance constructor method that is defined for objects of class Class. The GeneralNode may invoke the instance constructor method on its associated PersistentObject Class. The following steps are performed by this constructor method:

1. If the _constructor variable of the instance of Class is zero, the method creates a CodeLoader object and initializes the CodeLoader by passing it the name of the Class as an argument.

2. If the CodeLoader successfully loads the new class, the _constructor variable is assigned the address of a function that will invoke the static C++ constructor.

3. However, if the _constructor variable is still zero, the method fails by returning zero.

4. If the _constructor variable is non-zero, the method invokes the actual constructor. The return value of the constructor is also returned by the instance constructor method of the Class.

For more information on the operations performed by the CodeLoader and on adding code to running C++ systems, see [10, 7].

The organization of the containers of persistent objects is required to be space-efficient, see Section 3. Since there is no inherent limit on the length of Class names, storing one in the fixed-size structure of each GeneralNode is unacceptable. Instead, each GeneralContainer stores the names of the Classes of PersistentObjects that are associated with its GeneralNodes. This has several advantages:

- the Class names can be efficiently stored in a single buffer as null-terminated strings (so there is no need to impose an arbitrary limit on the length of Class names),
- if several GeneralNodes support the same Class, then the name only needs to be stored once, and
- each GeneralNode needs to store only an index (which requires only 1–4 bytes) into its container's list of Class names.

This section has discussed the typing of a persistent object store and the problems of dynamic code loading. In *Choices*, typing is accomplished by storing the name of the class of a persistent object in the container in which the object is stored. Invocation of a constructor for a new class

of persistent object is achieved using indirection. Objects of class Class represent the class of a persistent object and have an instance constructor method that may be used initialize the persistent object. Object-oriented techniques are used in *Choices* to simplify the programming of these mechanisms.

6 Naming

PersistentStores and PersistentObjects have both *logical* and *symbolic* names. The logical names of PersistentStores and PersistentObjects are unique identifiers, which are derived from the organization of nested PersistentStoreContainers:

- The method of identifying a PersistentStore depends on whether it is a Disk or a File.
  - A Disk is identified by a unique index within a computer system.
  - A File is identified by a pair comprising the identifier of its container and the File's index within its container.
- A PersistentObject is identified by the identifier of its underlying PersistentStore.

Within the Storage Management Layers of the framework, the file system identifies PersistentStores by their logical names, whereas higher layers and application programs identify PersistentStores by their symbolic names.

While logical names are based on the organization of nested PersistentStoreContainers, symbolic naming is orthogonal to organization. Therefore, although there is a single unique mapping of logical names to PersistentStores in a computer system, multiple symbolic name spaces can be mapped onto the set of logical names. Within each name space, multiple symbolic names can map to a single PersistentStore.

PersistentStores can be grouped into and given symbolic names by objects in both the Persistent Object and Object Interface Layers. In the Persistent Object Layer, PersistentStoreDictionaries, which are collections of <symbolic-key, logical-name> pairs, map symbolic names to the logical names of PersistentStores. Within any dictionary, the keys must be unique, but several keys may map to the same logical names. Using symbolic names, PersistentStores can be opened from, created in, added to, and removed from PersistentStoreDictionaries.

In the Object Interface Layer, FileSystemInterfaces use a set of PersistentStoreDictionaries to parse sequences of symbolic keys, called *pathnames*, and resolve them to logical names of PersistentObjects. FileSystemInterfaces can use various policies and mechanisms to unify the symbolic name spaces provided by PersistentStoreDictionaries. We have implemented a FileSystemInterface that provides the UNIX concept of root and current directories, mount tables, and symbolic links.

Since PersistentObjects can refer to other PersistentObjects directly, there is no requirement that they have symbolic names. It is advantageous for PersistentObjects that do not need to be identified by the users of applications to be given no symbolic names. In contrast, applications should be able to assign symbolic names to PersistentObjects both during and after their creation.

To assign a name to a PersistentObject as it is being created, applications can use the FileSystemInterface::create or PersistentStoreDictionary::create operations. Both operations take two arguments, the name of the object to be created and the Class to which it should belong. They both check to see if the object name already exists, and if it does not, they invoke the PersistentStoreContainer::create operation and store the object name in a PersistentStoreDictionary. The major difference between the two operations is that FileSystemInterfaces use path names while PersistentStoreDictionaries use single-element file names.

To assign a name to a PersistentObject after it has been created, applications can use the

PersistentObject::bindTo operation. This operation takes a FileSystemInterface and an object name as arguments, and it invokes an operation on the FileSystemInterface that causes an entry to be added to a PersistentStoreDictionary that contains the object name and refers to the PersistentObject.

7 Programming with PersistentObjects

The utility of persistent objects is directly related to how closely their usage resembles the usage of standard programming language objects. Many persistent object systems extend the base language to allow the almost seemless addition of persistence[1, 4, 3, 13]. We chose instead to use the inherent extensibility of C++ and to use a set of support classes and programmer conventions. Since we also use these classes and conventions to manage standard heap objects within the *Choices* kernel, programming with PersistentObjects is nearly identical to programming other objects in *Choices*.

The two classes that make programming with PersistentObjects virtually transparent to application programmers are: PersistentObjectStar and AutoloadPersistentObject. PersistentObjectStar and its subclasses are part of a "smart pointer" hierarchy that mirrors the major class hierarchy of *Choices*, which is rooted in class Object. The entire hierarchy of ObjectStars provide a type-safe mechanism for managing the garbage collection of heap objects. We originally developed ObjectStars (called "References" in [5]) for our first persistent object implementation. We generalized them to support all heap-allocated objects[10], because we found using them preferable to the explicit programming of reference-counting methods. ObjectStars provide a pointer-like syntax using the operator $->$ overloading feature of C++.

The AutoloadPersistentObject class lets programmers develop persistent object classes without explicitly programming the storage and retrieval of persistent data. When the PersistentStore::asA

operation is invoked, the PersistentStore checks to see if the requested Class of persistent object is a subclass of AutoloadPersistentObject. If it is, then the store loads the data for the PersistentObject after calling its constructor. When an AutoloadPersistentObject is no longer needed in primary memory, its noRemainingReferences operation is automatically invoked and the object's persistent data is written back to its PersistentStore before the object is deactivated.

Because our framework supports the data format of various existing file system standards, and because these formats are not amenable to being directly memory-mapped, the default behavior for PersistentObjects is that it requires programmers to manually implement the memory mapping. If one is designing data structures using C++ classes, then automatic memory mapping is feasible, and this behavior can be inherited from AutoloadPersistentObject. In either case, the choice of whether a subclass of PersistentObject implements or inherits the memory mapping is transparent to users of instances of the class.

8 Summary

In this paper we show that, although conventional operating systems do not provide support for efficient persistent object system implementations, one can build both conventional file systems and persistent object systems within the same persistent storage framework. We also demonstrated how the differing requirements of file and object storage can be accommodated as specializations of the abstract classes within the framework.

Conventional storage organization, storage management, and typing are too inefficient, restrictive, and inflexible to be used as the basis for a general persistent storage system. We have built a general persistent storage framework that uses nested containers of persistent data stores. The

stores may be customized for the types and sizes of persistent objects that they contain. The framework permits the selection of various garbage collection schemes to match the allowed sets of inter-object references. The types of the persistent objects that can be stored in the system may be dynamically extended.

We also discuss the naming and use of persistent objects within the *Choices* operating system. Persistent objects are not required to have symbolic names, but applications can give objects symbolic names as needed, both during and after creation. A set of support classes and programmer conventions enables nearly transparent usage of persistent objects.

References

[1] R. Agrawal and N. H. Gehani. Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++. In *Second International Workshop on Database Programming Languages*, Oregon Coast, June 1989.

[2] Lubomir Bic and Alan C. Shaw. *The Logical Design of Operating Systems.* Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[3] Robert Bretl, David Maier, Allen Otis, Jason Penney, Bruce Schuchardt, Jacob Stein, E. Harold Williams, and Monty Williams. The GemStone Data Management System. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 283–308. Addison-Wesley, Reading, Massachusetts, 1989.

[4] Alfred Leonard Brown. Persistent Object Stores. Technical Report Persistent Programming Report 71, Universities of St Andrews and Glasgow, October 1989.

[5] Roy H. Campbell and Peter W. Madany. Considerations of Persistence and Security in Choices, an Object-Oriented Operating System. In *International Workshop on Computer Architectures to Support Security and Persistence of Information*, pages 12.1–12.14, University of Bremen, Federal Republic of Germany, May 1990.

[6] Michael J. Carey, David J. DeWitt, Joel E. Richardson, and Eugene J. Shekita. Storage Management for Objects in EXODUS. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 341–370. Addison Wesley, Reading, Massachusetts, 1989.

[7] Sean M. Dorward, Ravi Sethi, and Jonathan E. Shopiro. Adding New Code to a Running C++ Program. In *Proceedings of the USENIX C++ Conference*, pages 279–292, San Francisco, California, April 1990.

[8] John A. Interrante and Mark A. Linton. Run-time Access to Type Information in C++. In *Proceedings of the USENIX C++ Conference*, pages 233–240, San Francisco, California, April 1990.

[9] Peter W. Madany. An Object-Oriented Approach towards A General Model of File Systems. Technical Report UIUCDCS-R-90-1607, University of Illinois at Urbana-Champaign, Dec 1990.

[10] Peter W. Madany, Roy H. Campbell, and Panogiotis Kougiouris. Experiences Building an Object-Oriented System in C++. In *Technology of Object-Oriented Languages and Systems Conference*, Paris, France, March 1991.

[11] José Alves Marques and Paulo Guedes. Extending the Operating System to Support an Object-Oriented Environment. In *Proceedings of OOPSLA '89*, pages 113–122, New Orleans,

Louisiana, September 1989.

[12] Peter Norton. *The Peter Norton Programmer's Guide to the IBM PC.* Microsoft Press, 1985.

[13] Joel E. Richardson, Michael J. Carey, and Daniel T. Schuh. The Design of the E Programming Language. Technical Report Computer Sciences 824, University of Wisconsin, Madison, February 1989.

[14] Vincent F. Russo, Peter W. Madany, and Roy H. Campbell. C++ and Operating Systems Performance: A Case Study. In *Proceedings of the USENIX C++ Conference*, San Francisco, California, April 1990.

[15] Marc Shapiro, Olivier Gruber, and David Plainfossé. A garbage detection protocol for a realistic distributed object-support system. Technical Report 1320, INRIA, November 1990.

[16] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Publishing Company, 1986.

[17] K. Thompson. Unix Implementation. *Bell System Technical Journal*, 57(6):1931–1946, July 1978.

To be presented at the 1991 International Workshop on Object-Orientation in Operating Systems, Palo Alto, CA, October 17-18, 1991.

NASA
58-61
93868
045139
21

Choices, Frameworks and Refinement*

Roy H. Campbell, Nayeem Islam, Ralph Johnson, Panos Kougiouris and Peter Madany
University of Illinois at Urbana-Champaign
Department of Computer Science
1304 W. Springfield Avenue
Urbana, IL 61801

Abstract

In this paper, we present a method for designing operating systems using object-oriented frameworks. A framework can be refined into subframeworks. Constraints specify the interactions between the subframeworks. We describe how we used object-oriented frameworks to design Choices, an object-oriented operating system.

1 Frameworks in an Object-Oriented Operating System

The design of *Choices*[1], an object-oriented operating system, comprises a hierarchy of frameworks. In the design, the concept of a framework subsumes the conventional organization of an operating system into layers. Frameworks not only allow the design of layers, but they also permit the construction of more complex structures. The use of frameworks permits design and code reuse and the consistent imposition of design constraints on all software, independent of the level at which it may be used.

The object-oriented operating system approach builds system software that models system resources and resource management as an organized collection of objects that encapsulate mechanisms, policies, algorithms, and data representations. A class defines a collection of objects that have identical behavior. Class hierarchies define relationships between classes that share common behavioral properties. Inheritance and inclusion polymorphism permit the methods of a concrete subclass to implement an invocation of a method on an abstract class. A framework of classes defines a design architecture that expresses the organization of an object-oriented implementation of a system. The framework can be refined into subframeworks, corresponding to the composition of a large complex system out of smaller interacting subsystems. A particular operating system implementation is just one of many possible ways that a framework for an operating system can be "instantiated."

Choices was designed from the beginning as an object-oriented operating system implemented in C++. The system runs stand-alone on the Sun SPARCstation II, Encore Multimax, Apple Macintosh IIx, IBM PS/2, and AT&T WGS-386. It supports distributed and shared memory multiprocessor applications, virtual memory, and has both conventional file systems and a persistent object store. The system has over 300 classes and 150,000 lines of source code.

In this paper we will describe how we have used the object-oriented notion of a framework in our work. *Choices* has the following frameworks: process management and exception handling, scheduling, synchronization, memory management, persistent storage, device management, name resolution, message passing, communication protocols, application interface, and instrumentation. We will discuss how particular frameworks in *Choices* have contributed to the organization of the system, techniques we have found helpful for building frameworks, and why frameworks are useful.

2 What is a Framework?

A framework is an architectural design for object-oriented systems. It describes the components of the system and the way they interact. In frameworks, classes define the components of the system. The interactions in the system are defined by constraints, inheritance, inclusion polymorphism, and informal rules of composition. *Choices* frameworks use class hierarchies to define single inheritance and C++ subtyping to express inclusion polymorphism. In practice, we

*This work was supported in part by NSF grant CISE-1-5-30035 and by NASA grants NSG1471 and NAG 1-163.

have found that the design of a complex system such as an operating system is best defined as a framework that guides the design of subframeworks for subsystems. The subframeworks refine the operating system framework, as it applies to a specific subsystem.

The framework for the system provides generalized components and constraints to which the specialized subframeworks conform. The subframeworks introduce additional components and constraints and subclass some of the components of the framework. Recursively, these subframeworks may be refined to further frameworks. Frameworks simplify the construction of a family of related systems by providing an architectural design that has common components and interactions. An instance of a framework is a particular member of the family of systems.

Frameworks both support and augment the traditional layered approach that has been used to design operating systems. In both approaches the problem domain is divided into smaller domains. A layer represents an abstract machine that hides machine dependencies and provides new services. The abstract machine is presented as a set of subroutines. A framework introduces classes of components that encapsulate machine dependencies and define new services. A layer introduces an interface between implementations that is constrained by the set of calls that are defined.

A framework defines interfaces in the form of the public methods of abstract classes. It imposes restrictions on the implementation of an interface by the constraints it imposes. In the layered approach, the design of each layer is independent. Algorithms or data structures in one level may be similar to those in other levels, but the level approach to design has no way to express that similarity. Instead, a framework may have several different instantiations and implementations within a system; it may be reused. The constraints of a framework allow more complex interactions than between levels. The framework approach subsumes the layered approach because the basic properties of the layered approach can be modeled by frameworks. However, the framework approach also allows the constraints within a particular layer to be expressed. Finally, a framework can be defined in terms of abstract classes that are bound to specific concrete classes at run-time using inheritance and inclusion polymorphism. This provides the compile time independence that is exhibited by, for example, the application interface layer but also allows dynamic binding as, for example, is necessary to allow device drivers to be added or changed in a running system.

3 Choices Frameworks

The framework for *Choices* introduces abstract classes that represent the high-level concepts that are fundamental to an operating system. Subframeworks are defined that refine these concepts in the context of a subsystem of the operating system. The *Choices* framework provides the subframeworks with the design consistency constraints that are required to link them together into a system. The *Choices* framework consists of three abstract classes: *MemoryObject*, *Process*, and *Domain*.

In this paper, we describe the Persistent Storage, Device Management and Message Passing subframeworks. The virtual memory subframework has already been described by Johnson and Russo [5]. There are three stages to building a *Choices* subsystem. First, the abstract properties of the subsystem are collected. Next, a subframework is developed which is consistent with the *Choices* framework and which encodes the properties of the subsystem. This subframework defines classes and constraints. Finally, the subsystem is created as an implementation of the subframework, with concrete classes specializing the abstract classes of the framework and instances that conform to the constraints specified by the framework. The following diagram shows the development of a subsystem from abstract properties to a concrete implementation.

$$\textit{Abstract Properties} \rightarrow \textit{subframework} = \{\textit{classes}, \textit{constraints}\}$$

$$\rightarrow \textit{subsystem} = \{\textit{instance of subframework}\}$$

3.1 Choices subframeworks

Persistent Storage The *Choices* persistent storage framework[7] introduces a hierarchy of classes that can be combined to build both standard and customized storage systems. It is flexible enough to support both persistent storage systems and traditional file systems efficiently[8]. The framework abstracts five major properties of persistent storage systems:

- Persistent Storage,
- Storage Organization and Device Sharing,
- Naming,
- Data Structuring, and
- Application Interface.

The persistent storage framework divides a persistent storage system into three layers and is, therefore,

an example of a framework that subsumes a traditional layering structure. The top layer contains objects that present application interfaces, the middle layer contains objects that name files and structure the data within files, and the bottom layer contains objects that store and organize persistent data. The bottom layer can be further divided into several levels.

The *PersistentStore* class, which is a subclass of *MemoryObject*, abstracts persistent storage by defining persistent data stores which store and retrieve blocks of data using a random access method. Concrete *PersistentStores* are categorized by the two subclasses of *PersistentStore*: *Disk* and *File*. *Disks* encapsulate physical storage devices like hard disk drives, floppy disk drives, and RAM disks. They communicate with objects in the I/O subsystem. *Files* encapsulate logical storage devices like UNIX inodes and disk partitions. Each file has a source *PersistentStore* that supplies it with data from a lower level of the file system. *Files* provide a *window* into their source *PersistentStore*. The size of this window can be fixed or variable and can range from zero up to the size of the source *PersistentStore*. The window can be contiguous or divided into discontiguous regions of blocks.

The *PersistentObject* class defines persistent objects, which encapsulate and provide operations on the data managed by a persistent store. While a *PersistentStore* provides random access to an uninterpreted sequence of data, a *PersistentObject* interprets the data of a persistent store as having a format. Each *PersistentStore* has an associated *PersistentObject* class that provides a data abstraction and encapsulation of the persistent data in the store. At runtime, there is a one-to-one correspondence between an instance of a *PersistentStore* and its associated *PersistentObject*. The *PersistentStore* thus provides the underlying data for its associated *PersistentObject*. Subclasses of *PersistentObject* abstract the organization, naming, and data structuring properties of the persistent storage framework.

Storage is organized and storage devices are shared by dividing *PersistentStores* into nested levels of smaller *PersistentStores*. The *PersistentStoreContainer* class defines objects that divide a persistent store into an indexed collection of smaller stores (i.e. a collection of *Files*). *PersistentStoreContainers* that contain variable-length files, or ones that can create new files, use a *BlockAllocator* to manage the allocation of data blocks for the files.

Naming is orthogonal to storage organization. The *PersistentStoreDictionary* class defines objects that map symbolic names to the indices used by *PersistentStoreContainers*. While *Files* must be contained in exactly one container, they can be named by several dictionaries.

The framework incorporates three models for structuring the data within files: as arrays of bytes or words (defined by subclasses of *PersistentArray*), as collections of records (defined by subclasses of *RecordFile*), and as data structures encapsulated by persistent objects (defined by subclasses of *AutoloadPersistentObject*). The first model is suited to the C programming language and the UNIX and MS-DOS operating systems. The file system presents a random-access interface to sequences of bytes and imposes no additional structure. The second model fits programming languages like Cobol, PL/1, and Pascal and operating systems like VMS. The file system presents data as records that can correspond to the types of data structures of the language. The third model fits programming languages like C++ and object-oriented operating systems like *Choices*. The object storage system presents data as objects that are instances of user-defined subclasses of *AutoloadPersistentObject*.

The interfaces provided by the naming and data structuring classes are abstract enough to be used directly by application programs; but conventional file systems commonly define an additional layer of abstraction between files and application programs. The *FileSystemInterface* class provides this extra layer by organizing the dictionaries from various containers into a single hierarchy. Subclasses of the *RecordStream* class provide stream-oriented application interfaces for both *PersistentArrays* and *RecordFiles*.

Device Management The device management framework in *Choices* abstracts three properties of a typical I/O architecture:

- I/O Devices (*Device*)
- I/O Controllers (*DevicesController*)
- Addition and removal of devices, controllers and drivers(*DevicesManager*)

The device management framework[6] in *Choices* consists of two hierarchies and a DevicesManager class. One hierarchy is based on the abstract class *Device*. The other hierarchy is based on the abstract class *DevicesController*. An instance of a *Device* is constructed and bound to the NameServer when it is added to the system. Each *Device* acts as a server for components of other *Choices* frameworks. For instance, a DiskDevice acts as a server of classes of the file system framework. In turn, most of the *Devices* act as clients of

DevicesController objects. For instance, two DiskDevices representing disks attached to the same hardware controller act as clients of the same *DiskController*. The protocol of a *Device* depends on the physical device it represents. For instance, DiskDevices have `::read()` and `::write()` methods that transfer a number of disk blocks. SerialLineDevices have Input/Output methods to transfer strings of characters and control methods like `::setBaudRate()` to set control parameters.

Instances of *DevicesControllers* classes represent hardware controllers. A *DevicesController* acts as a server for possibly several *Devices*. A *DevicesController* is not visible to the user of a device. I/O operations should only be requested from a *Device*. The only other framework that interacts with a *DevicesController* is the Exception Handling framework. The interface between a *Device* and a *DevicesController* is a message interface based on *Command* objects. User requests on *Devices* cause the construction of one or more *Commands* which are then sent to a *DevicesController* object using the `::sendCommand()` method. A message interface between the *DevicesController* and the *Device* has two advantages. The first is that a *Device* can be reused with different *DevicesControllers*. For example, a *DiskDevice* can be used as the *Device* of a machine-dependent DiskController and a machine-independent SCSIDiskController. The second advantage of the message interface is that it does not force a *DevicesController* to have a specific interface that depends on its devices. The protocol of a *DevicesController* subclass can change without requiring a change to existing *Devices*. On the other hand, the message interface cannot be checked at compile-time to ensure consistency. We think that this is not a major problem, since the interface is internal to the framework.

The DevicesManager class is the third component of the framework. Each system has only one object of this class. When a *DevicesController* is loaded into the system it registers with the DevicesManager object. Hardware controllers and devices that are added to the system are also registered with the DevicesManager. The DevicesManager is informed of the addition and removal of hardware components by the system administrator or by the cooperation of hardware and machine-dependent software. The DevicesManager matches physical controllers with registered *DevicesControllers*. For each physical controller a "matching" *DevicesController* is instantiated. In addition, for each physical device a *Device* is constructed and returned when the method `DevicesController::attachDevice()` is invoked. The new *Device* is then bound to the NameServer.

Other frameworks use the Device Management framework with the help of classes that provide communication between the two frameworks. For instance, a Disk is a *PersistentStore* that does Input/Output using a DiskDevice. *Devices* are converted to objects in other hierarchies using the *Choices* conversion mechanism. Conversion is a term introduced in Smalltalk[2] and used for the collection classes. We generalize the conversion mechanism to apply to any class. We also combined the conversion mechanism with double dispatching[3] so that new interframework classes can be added to the system without changing existing classes inside the frameworks.

Message Passing This section describes a subframework for message passing designed to support hypercube applications on a shared memory machine. The abstract properties of message passing subframework can be categorized into five parts, with various options, as shown below.

- *Location* of Message system:
  - *User space*
  - *Kernel*
- *Message* Semantics and Process control:
  - *Hypercube*
- *Transport*:
  - *Process:* an independent process copies the message from source to destination.
  - *Buffered:* the receiver process incurs the overhead of message transfer.
- *Synchronization*:
  - *Semaphore*
  - *Spinlock*
- *Transfer* of the exchange of data implemented by:
  - *DoubleCopy:* the message is copied into and out of the kernel.
  - *SingleCopy:* the message is copied once from sender buffer to receiver buffer.
  - *PointerTransfer:* exchange of buffer pointers with no copy of data.

From the abstract properties we can derive a set of *abstract classes* and their associated hierarchies. The hierarchies may express some of the constraints in their names: other constraints are described in the

description of the interaction of the classes. These abstract classes are *MessageTransport*, *MessageSemantics*, *MessageSynchronization*, and *MessageTransfer*. Each of these have concrete subclasses that implement the abstractions defined by them. In the following paragraphs we will only describe a representative sample of concrete subclasses. The *KernelMessageSystem* and *UserMessageSystem* classes define, as well as implement, the location semantics. A more complete description of the message passing subframework can be found in Islam and Campbell [4].

The *MessageSemantics* class defines the semantics of message passing, such as whether asynchronous and synchronous message passing is supported. It also defines the process control semantics such as whether processes should be gang scheduled for the application.

The *MessageTransport* class defines whether a separate process will be delivering the message or whether the receiving process will incur the overhead of the transfer mechanism. The *MessageTransfer* class defines the management of message queues and how the data is actually moved between sender and receiver.

The *MessageSynchronization* class defines the message synchronization semantics. The concrete subclasses SpinLockBufferedMessageSynchronization, SemaphoreBufferedMessageSynchronization, SpinLockUserMessageSynchronization and SemaphoreProcessMessageSynchronization provide various implementations of this concept. The type of synchronization employed is used as a prefix to the concrete class. Since there is a restriction on how transport and synchronization may be mixed, the type of transport is also factored into the class: for example, the SpinLockBufferedMessageSynchronization class assumes a buffered transport mechanism with spin-locks for synchronization. MessageSingleTransfer, MessageDoubleTransfer, MessageUserTransfer and MessagePointerTransfer implement various schemes for moving a message from sender to receiver. For example, MessageDoubleTransfer copies a message from the sender's address space into the kernel and then from the kernel into the receiver's address space. Given the above framework it is possible to create a specific message passing system. For example, a collection of instances of the classes, KernelMessageSystem, BufferedMessageTransport, BufferedSpinLockMessageSynchronization, and MessageDoubleTransfer defines a hypercube style message passing system that is kernel based, lets the receiver process incur the cost of message transfer, provides process synchronization through spin-locks and copies the message into the kernel domain and then into the user address space (double copy semantics).

The message passing subframework makes reference to the Process framework class in its *MessageSemantics* abstract class. In addition, it references the MemoryObject framework class in its *MessageTransfer* abstract class.

The class hierarchies described above have been reached though iterative improvement. A concept was tried and when it did not work it was modified. One of the most important aspects of the design is *reuse*. For example, it is possible to combine a SpinLockMessageSynchronization class with either a MessageSingleTransfer to MessageDoubleTransfer. A earlier design merged the transfer and the synchronization hierarchies. This was clearly a mistake as the synchronization and transfer mechanisms are separate concepts. The old design forced the transfer mechanism to be replicated for both the semaphore and spin-lock modes of synchronization. Keeping these separate allows one instance of the transport mechanism to be used with several synchronization mechanisms.

4 Advantages of Frameworks

In this section, we describe some of the major advantages of using frameworks for designing an operating system. We demonstrate the advantages with examples from the *Choices* operating system.

- *Code Reuse* is normally achieved through the reuse of existing components and through polymorphism. With frameworks, code can also be reused through inheritance. The use of virtual functions in C++ for example, allows large bodies of code to be reused. In the persistent storage subframework, several abstract classes, including *PersistentStoreContainer* and *PersistentStoreDictionary*, implement all public operations. These operations are defined using several simple operations that subclasses must implement.

- *Design Reuse* is achieved in frameworks by reusing abstract concepts from one subframework in another framework. For example, the notion of MemoryObjects may be used in the persistent store subframework as well as in the virtual memory subframework. Frameworks allow this commonality to be described and reused.

- *Portability* is achieved in frameworks by separating machine-dependent parts of design from

the machine-independent parts. For example, an abstract class may have implementations of the machine-independent parts of a component, but machine-dependent parts will be specified by pure virtual functions that must be supplied by a subclass. For example, there is a CPU class that is machine-independent but it has concrete subclasses that are tailored to the SPARC, i386, NS32332, and MC68030 processors.

- *Rapid Prototyping* of different concepts is possible in frameworks because it supports code and design reuse. Code reuse and design reuse reduce coding time and design time, respectively. Once an abstract class has been built, it is only necessary to supply implementations of its pure virtual functions in a concrete class. For example, we were able to compare and contrast several message delivery mechanisms in the *Choices* message passing subframework.

- It is possible to customize for *performance*. For example, in the message passing subframework we allow synchronization through semaphores and spin-locks. For hypercube applications, the spin-lock version is a faster synchronization mechanism.

5 Techniques for building frameworks

In this section, we identify and describe some useful techniques for implementing frameworks. We provide example uses from *Choices*.

- *Abstract* classes provide generalized interfaces for *concrete* classes. *Concrete* classes are implementations of *abstract* classes. In *Choices*, the persistent storage framework involving PersistentStores, PersistentStoreContainers, and PersistentStoreDictionaries is used to introduce constraints on the partitioning of disks, provision of logical files, and implementation of file named.

- *Inclusion Polymorphism* refers to a subclass being a subtype of a superclass. This allows a subclass to be used wherever a superclass is expected. In *Choices*, all devices and device controllers are derived from a device-driver framework. Any device written to use an abstract controller interface may use any instantiation of a controller such as a SCSI bus controller. Further, given a request for a particular implementation of an I/O interface, the system is free to bind that request to any convenient implementation of the interface provided that the class of the object requested and class of the service offered satisfies the subtyping requirement.

- *Constraints* are descriptions of relationships between abstract classes of frameworks or the relationship between concrete and abstract classes within a framework. The use of constraints is most evident in how instances of concrete classes are combined. For example, in the message passing subframework a buffered message interface must be used with a buffered message delivery mechanism. It cannot be combined with a process message delivery mechanism.

- *Dynamic code loading* allows one to specify an abstract class when the system is designed and add a concrete descendant class of the abstract class at run-time. For example, a *Choices* device driver consists of a *DevicesController* class and a number of *Device* classes. A new device driver can be added to the system by loading the concrete subclasses of the *DevicesController* and the *Device* classes that form the device driver.

- *Delayed Binding* is the ability to determine dynamically the methods to which an object responds (often referred to as the signature of the object). In object-oriented systems this is not known until run-time. In C++, delayed binding is a result of using virtual functions. All abstract classes in *Choices* use virtual functions.

- *Conversion* allows objects to be changed at run-time into other objects. Conversion does not modify the original object; instead, a new one is created using the data of the old object. Subclasses of ProxiableObject implement the conversion process by responding to the `asA` message[7]. It takes an argument that may be the name of either a concrete or an abstract class and returns a reference either to an instance of the argument or an instance of a concrete subclass of the argument, respectively. The the `asA` method uses the `supports` method to ensure that the underlying data is compatible with the given class. For example, in the *Choices* device management subsystem a serial line can be converted to an input stream and an output stream. In the persistent storage system, a persistent store can be converted into a persistent object.

6 Conclusions

Our experience has shown that an object-oriented framework is an effective technique for designing a complex software system such as an operating system. In this paper, we have shown how complicated components of the operating system can be designed and the interfaces between the different components defined using frameworks. We also show how a framework for a system can be used to help design the subframeworks required for subsystems of the system. Parts of the framework are refined and specialized for the subframework. There are, however, critical parts of a framework that have only informal definition. In particular, we found that a suitable notation for expressing many of the informal constraints between components of a system is lacking. The relationships that can be expressed by the classes in C++ was insufficient to express all the constraints that accompanied the design of the *Choices* frameworks.

References

[1] Roy H. Campbell, Vincent Russo, and Gary Johnston. Choices: The Design of a Multiprocessor Operating System. In *Proceedings of the USENIX C++ Workshop*, pages 109–123, Santa Fe, New Mexico, November 1987.

[2] Adele Goldberg and David Robson. *Smalltalk-80: The Language.* Addison-Wesley, Reading, Massachusetts, 1989.

[3] Kurt J. Hebel and Ralph E. Johnson. Arithmetic and double dispatching in Smalltalk-80. *Journal of Object Oriented Programming*, March/April 1990.

[4] Nayeem Islam and Roy Campbell. The performance of message based applications on an object oriented operating system. Technical Report UIUCDCS-R-88-1455, University of Illinois Urbana-Champaign, April 1991.

[5] Ralph E. Johnson and Vincent F. Russo. Reusing object-oriented designs. Technical Report UIUCDCS-R-91-1696, University of Illinois at Urbana-Champaign, May 1991.

[6] Panagiotis Kougiouris. A Device Management Framework for an Object-Oriented Operating System. Master's thesis, University of Illinois at Urbana-Champaign, August 1991.

[7] Peter W. Madany. An Object-Oriented Approach towards A General Model of File Systems. Technical Report UIUCDCS-R-90-1607, University of Illinois at Urbana-Champaign, Dec 1990.

[8] Vincent F. Russo, Peter W. Madany, and Roy H. Campbell. C++ and Operating Systems Performance: A Case Study. In *Proceedings of the USENIX C++ Conference*, San Francisco, California, April 1990.

Proceedings of the Second International Conference.
San Mateo, CA: Morgan Kaufmann, 1991.

A General Framework for Modal Deduction

Sg-61
93869
10p.
045171

Alan M. Frisch and Richard B. Scherl
University of Illinois
Department of Computer Science and Beckman Institute
405 North Mathews
Urbana, Illinois 61801
frisch@cs.uiuc.edu scherl@cs.uiuc.edu

Abstract

A general method of automated modal logic theorem proving is discussed and illustrated. This method is based on the substitutional framework for the development of systems for hybrid reasoning. Sentences in modal logic are translated into a constraint logic in which the constraints represent the connections between worlds in the possible world semantics for modal logic. Deduction in the constraint logic is performed by a non-modal deductive system which has been systematically enhanced with special-purpose constraint processing mechanisms. The result is a modal logic theorem prover, whose soundness and completeness is an immediate consequence of the correctness of the non-modal deductive system and some general results on constraint deduction. The framework achieves significant generality in that it provides for the extension of a wide range of non-modal systems to corresponding modal systems and that this can be done for a wide range of modal logics.

1 Introduction

A wide range of modal logics have been finding applications in many areas of AI research and recently a number of automated deduction methods for modal logics have been developed. A number of these methods utilize ordinary non-modal inference rules augmented with a special purpose unification procedure that reasons about the accessibility of possible worlds. The deduction methods that fall into this class have been based on a variety of non-modal methods including resolution [Ohlbach, 1988, Auffray and Enjalbert, 1989], sequent calculus [Jackson and Reichgelt, 1989] and the matrix method [Wallen, 1990]. Additionally, outside of the methods designed specifically for automation, Fitting[1983] has developed a related approach based on deductive tableaux. This leads one to wonder whether there are any methods for systematically constructing modal deduction systems from non-modal systems, whether an understanding of the modal systems can be obtained from an understanding of the original non-modal systems, and whether there is a common set of principles underlying systems constructed in this manner.

This paper presents a framework for the *systematic* design of modal deduction systems that allows us to respond affirmatively to each of the above questions. The framework achieves significant generality in that it provides for the extension of a wide range of non-modal deduction systems to corresponding deduction systems for a wide range of modal logics. Along with this transformation of the deductive rules, the proofs of completeness of the original deductive system are transformed into a proof of completeness of the new deductive system. The resulting description and proof of correctness of the new system is relatively simple because it draws upon the general principles of the framework. The framework enables us to provide a proof of completeness of a deductive method, based upon resolution and paramodulation, for modal logics with equality. Most of the features of most of the members of the class of methods mentioned above can be recast into this framework. Those aspects of the methods that can not be so reconstructed serve to delimit the domain of applicability of the framework. A deeper understanding of how this class of related approaches work has been achieved.

Very roughly stated, our method works as follows: We start with a set of modal sentences on which the deduction is to be performed. These sentences are translated into non-modal sentences that contain K-atoms, atoms whose predicate symbol represents the accessibility relation between possible worlds. In the case of varying domain modal logics, E-atoms, atoms whose predicate symbol denotes the existence

Principles of Knowledge Representation and Reasoning: Proceedings of the Second International Conference.
San Mateo, CA: Morgan Kaufmann, 1991.
(Slightly revised version)

Generalizing Atoms in Constraint Logic

510-61
93870
24514
13 p.

C. David Page Jr. and Alan M. Frisch
Dept. of Computer Science and Beckman Institute
University of Illinois
405 North Mathews Ave.
Urbana, IL 61801

Abstract

This paper studies the generalization of atomic formulas, or atoms, that are augmented with constraints on or among their terms. The atoms may also be viewed as definite clauses whose antecedents express the constraints. Atoms are generalized relative to a body of background information about the constraints. The paper first examines generalization of atoms with only monadic constraints. The paper develops an algorithm for the generalization task and discusses algorithm complexity. It then extends the algorithm to apply to atoms with constraints of arbitrary arity. The paper also presents semantic properties of the generalizations computed by the algorithms, making the algorithms applicable to such problems as abduction, induction, and knowledge base vivification. The paper emphasizes the application to induction and presents a pac-learning result for constrained atoms.

1 Introduction

Unification plays an important role in many automated deduction systems. It comes in a number of forms including term unification, string unification, unification with built-in equality, sorted unification, and feature structure unification, as well as variations of these. The common feature of all these forms of unification is that they compute maximal lower bounds in some partially-ordered set of syntactic objects. Turning unification on its head yields an operation called "generalization," or "anti-unification," that computes minimal upper bounds. As with unification, one can imagine many forms of generalization. We conjecture that generalization will play a role in non-deductive reasoning (e.g., abduction, induction and analogy) as important as that played by unification in deductive reasoning.

Section 2 of the paper reviews ordinary generalization, which has been studied by Plotkin [1970; 1971], Reynolds [1970], Lassez, Maher and Marriott [1988], and Lassez and Marriott [1987]. Ordinary generalization is a purely syntactic operation that operates solely on the basis of expression structure. Section 2 also reviews sorted generalization [Frisch and Page, 1990], an operation that acts on atoms whose variables are constrained by associated sorts, or monadic constraints. Sorted generalization is governed by the logical consequences of a taxonomic theory that describes the sorts. Section 3 first extends the generalization operation to allow monadic constraints on arbitrary terms in the atoms, rather than on variables only. Section 3 extends the generalization operation further still, so that it operates on atoms with constraints of arbitrary arity among their terms. The resulting operation acts with respect to a theory that describes the constraints.

Elsewhere [Frisch and Page, 1990] we examine applications of sorted generalization to abduction, induction, knowledge base vivification, and analogical reasoning. These applications are based on semantic properties of sorted generalization. Section 4 of this paper shows that constraint generalization has analogous semantic properties. Section 4 then concentrates on the application to induction, discussing polynomial-time pac-learning of constrained atoms and polynomial-time learning of constrained atoms by equivalence queries.

From a constraint logic programming point of view, an atomic formula, A, with attached constraints can be seen as a definite clause whose head is A and whose body contains zero or more constraint literals but no ordinary literals. Alternatively, such a formula can be seen as a definite clause of ordinary logic whose head is A and whose body is a conjunction of all the constraints. Thus, our work is related to Buntine's [1988] work on generalized subsumption. Section 5 examines this relationship.

6 Conclusion

A general framework for constructing and proving the correctness of methods for automated deduction in modal logic has been discussed and illustrated. It provides a systematic method for converting a non-modal deduction procedure and its proof of correctness to a corresponding modal deduction procedure with a proof of correctness. The framework achieves significant generality in that it provides for the extension of a wide range of non-modal systems to corresponding modal systems and that this can be done for a wide range of modal logics. The ease with which we were able to add equality to the framework speaks for its usefulness.

Most of the features of the members of the general class of related methods [Wallen, 1990, Ohlbach, 1988, Jackson and Reichgelt, 1989, Auffray and Enjalbert, 1989] can be recast as particular instantiations of our general framework. The approach developed here has the advantage of producing relatively simple proofs of soundness and completeness. They follow from general results on substitutional reasoning. A deeper understanding of how these related approaches work has been obtained. Also, the general framework has provided a common set of principles which relate various modal and non-modal deduction methods.

Many of the related methods are limited to serial modal logics. Ohlbach adds an additional inference rule in order to handle the non-serial case. Wallen, using a version of the matrix method which does not require prenexing, is able to handle non-serial logics without any additional machinery. Our framework explains why non-seriality is a problem. The framework specifies when a simple modification of an ordinary inference rule that works on prenex Skolem normal form formulas suffices to maintain completeness in the constrained case. Non-serial logics fall outside the boundary as the constrained sentences cannot be put in equivalent prenex form.

On the other hand, logics for which it is necessary to add the Euclidian axiom to Σ are not considered by the other researchers. They fit readily into the overall framework, but the difficulty lies in extending the method of sequence unification to a test for Σ-solvability when Σ contains the Euclidian axiom.

The approach has led to a number of problems, which are all topics of our current and future research. These include finding a representation for constraints that avoids generating multiple maximally general unifiers, developing a procedure for testing for Σ-solvability in the presence of the Euclidian axiom, extending the framework to cover inference rules that work on non-prenex formulas, and also extenting the method to a wider class of modal and intensional logics.

References

[Auffray and Enjalbert, 1989] Yves Auffray and Patrice Enjalbert. Modal theorem proving: An equational viewpoint. In *IJCAI-89*, pages 441–445, Detroit, August 1989.

[Buntine, 1988] Wray Buntine. Generalized subsumption and its applications to induction and redundancy. *Artificial Intelligence*, 36(2):149–176, September 1988.

[Fitting, 1983] Melvin C. Fitting. *Proof Methods for Modal and Intuitionist Logics.* D. Reidel Publishing Co., Dordrecht, 1983.

[Frisch, 1991] Alan M. Frisch. The substitutional framework for sorted deduction: Fundamental results on hybrid reasoning. *Artificial Intelligence*, 1991. (to appear, an earlier version appeared in KR89).

[Genesereth and Nilsson, 1987] Michael R. Genesereth and Nils J. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kaufman, Los Altos, California, 1987.

[Jackson and Reichgelt, 1989] Peter Jackson and Han Reichgelt. A general proof method for modal predicate logic. In *Logic-Based Knowledge Representation*, chapter 8, pages 177–228. MIT Press, Cambridge, Massachusetts, 1989.

[Jaffar and Lassez, 1987] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In POPL-87, pages 111–119, Munich, January 1987.

[Kripke, 1963] S. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

[Moore, 1980] Robert Moore. Reasoning about knowledge and action. Technical Note 191, SRI International, October 1980.

[Morgan, 1976] C.G. Morgan. Methods for automated theorem proving in nonclassical logics. *IEEE Transactions on Computers*, C-25:852–862, 1976.

[Ohlbach, 1988] Hans Jürgen Ohlbach. A resolution calculus for modal logic. In E. Lusk and R. Overbeek, editors, *CADE-9*, volume 310 of *Lecture Notes in Computer Science*, pages 500–516. Springer-Verlag, Berlin, May 1988.

[Van Hentenryk, 1989] Pascal Van Hentenryk. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, MA, 1989.

[Wallen, 1990] Lincoln A. Wallen. *Automated Proof Search in Non-Classical Logics: Efficient Matrix Proof Methods for Modal and Intuitionistic Logics.* The MIT Press, Cambridge, Massachusetts, 1990.

case of (single modality) $S5$, the number of unifiers can be limited to 1 by exploiting the fact that the worlds form equivalence classes and every world is connected to every other world in its equivalence class.

The correctness of the sequence unification procedure as a method for testing Σ-solvability of constraints is established by the following theorem.

Theorem 10 *Given a variable disjoint L_1, C_1 and L_2, C_2 and Σ, the sequence unification procedure succeeds if and only if L_1 and L_2 have a most general unifier θ and the constraint $C = (C_1' \wedge C_2')\theta$ is Σ-solvable—i.e. $\Sigma \models \exists (C_1 \wedge C_2)\theta$, where C_1' and C_2' are the K-atom representations of the sets of sequences C_1 and C_2. When successful, the final result of this procedure is the sequence representation of a constraint equivalent to $(C_1' \wedge C_2')\theta$.*

The sequence unification procedures found in [Wallen, 1990, Ohlbach, 1988, Jackson and Reichgelt, 1989, Auffray and Enjalbert, 1989] can be shown to be variants of the procedure laid out above.

5 Varying Domain Logics

In constant domain modal logics, the same entities exist at all possible worlds. Both the Barcan formula (41) and the converse Barcan formula (42) are theorems.

(41) $\forall x \; \Box \; A(x) \rightarrow \Box \; \forall x \; A(x)$
(42) $\Box \; \forall x \; A(x) \rightarrow \forall x \; \Box x \; A(x)$

In the varying domain case neither formula may be a theorem or either one of the two (but not both) may be a theorem.

In translating varying domain modal logics into a reified modal logic, it is necessary to introduce a binary predicate E as a constraint atom. For example $\forall x \, P(x)$ is translated into $\forall x{:}E(x, 0) \; P(x, 0)$ which can be read as saying for all entities x that exist in the initial world, $p(x)$ is true in the initial world. In general, the atom $E(x, w_i)$ restricts the variable x to range over elements that exist in world w_i. The quantifier rules in the translation function $\mathbf{T_i}$ are replaced with the following:

- $\mathbf{T_i}(\forall x \varphi) = \forall x'{:}E(x', w_i) \; \mathbf{T_i}(\varphi)$
- $\mathbf{T_i}(\exists x \varphi) = \exists x'{:}E(x', w_i) \; \mathbf{T_i}(\varphi)$

It is also necessary to add $\forall w \; \exists x \; E(x, w)$ to the constraint theory to ensure that the domains in each world are non-empty. If the Barcan formula is to be a theorem, it is necessary to add

(43) $\forall w_i, w_j \, (K(w_i, w_j) \wedge \forall x \; E(x, w_j)) \rightarrow E(x, w_i)$

to the constraint theory. Likewise, if the converse Barcan formula is to be a theorem of the logic,

(44) $\forall w_i, w_j \, (K(w_i, w_j) \wedge \forall x \; E(x, w_i)) \rightarrow E(x, w_j)$

needs to be added.

In the varying domain case both K and E are constraint atoms. The transformation to Skolem normal form and the results of Section 3 are general enough to apply when there are more than one constraint atoms.

Consider a varying domain logic with no additional axioms. Let's try to prove (42), the converse Barcan formula. The translation of (42) is:

(45) $\exists w_1{:}K(0, w_1) \; \exists x{:}E(x, w_1) \; \neg A(x, w_1) \; \vee$
$\forall x{:}E(x, w_1) \; \forall w_1{:}K(0, w_1) \; A(x, w_1)$

When the formula is negated and placed in clausal form, the result is two clauses as follows:

(46) $A(x, w_1)/K(0, w_1) \wedge E(x, w_1)$
(47) $\neg A(f, sk)/\emptyset$

The constraint theory for this problem contains $E(f, 0)$ and $K(0, sk)$.

Note that there is a potential resolvent of (46) and (47). But it is then necessary to show that $\Sigma \models E(f, sk)$. Since $\Sigma \not\models E(f, sk)$, this step of resolution is not possible and the proof of (42) fails. But if (44) were added to Σ, (46) and (47) could resolve to yield (48).

(48) $\Box / K(0, sk) \wedge E(f, sk)$

Now it is the case that $\Sigma \models K(0, sk) \wedge E(f, sk)$ and a proof of (42) is obtained.

The constraint solving procedure of Section 4 can be extended to the varying domain case following the method utilized by Wallen[1990]. The basic idea here is that the sequence representation of the constraints is augmented with a table that associates each term that occurs in the first argument position of the E atoms (in the K and E-atom representation of the constraints) with the sequence terminating in the second argument position of the E atom. In the case of varying domain logics (without (43) and (44) in the K and E-atom representation of Σ), the substitution of term t_1 for t_2 requires a check to see if $UNIFY(s(t_1), s(t_2))$ succeeds where $s(t)$ is the sequence associated with the term t. If for example the converse Barcan formula is to be a theorem of the logic (i.e. (44) is in the K and E-atom representation of Σ), it is necessary to verify that either $UNIFY(s(t_1), s(t_2))$ or $UNIFY(s'(t_1), s(t_2))$ succeeds where $s'(t_1)$ is some subsequence of $s(t_1)$.

We reserve the word *path* for a list, beginning with 0, of vertices of the graph representation of the least Herbrand model of Σ while the *sequences* are in the language. Both are lists of symbols that begin with the constant 0. The paths contain only ground terms, while the sequences may contain variables as well. Since the graph of the connectivity relations forms a tree rooted at 0, there can only be one path to any term in the graph. This is a property of the graph representation of the least Herbrand model of any constraint theory Σ from a reified set of sentences of the modal logic D. We may think of these terms as worlds and the paths as accessibility relations in a Kripke structure.

The basic idea here is that since there can be only one path to any ground term in the minimal Herbrand model of Σ, a constraint which specifies that there must be two paths (ground sequences corresponding to two paths) to a term cannot be Σ-solvable. Unification determines whether or not there are ground sequences (corresponding to paths in the least Herbrand model) in the intersection of two sequences. If unification succeeds, the result will be an expression which represents the set of ground sequences in the intersection of the two original sequences.

Consider the sequence representation of Σ_{2wm}. The set contains just two sequences—$\langle 0 \stackrel{K_A}{\rightarrow} Sk_2 \rangle$ and $\langle 0 \stackrel{K_A}{\rightarrow} w_1 \stackrel{K_B}{\rightarrow} Sk(w_1) \rangle$. When reasoning over the minimal Herbrand model of Σ_{2wm}, the only ground terms (worlds) we know about are 0, Sk_2 and those terms with Sk as the outermost function symbol. The only connectivity relations we know about are those summarized by the sequences. There is only one path from 0 to any of the other ground terms from the Herbrand universe.

As an illustration of the deductive process with the sequence representation of constraints, consider the refutation for the two wise-man puzzle when represented with clauses (34) – (37) and the sequence representation of Σ_{2wm}. At each step, the sequence unification procedure is given a set of sequences from each parent. In this simple problem, the procedure *UNIFY* is only called once by each invocation of the sequence unification procedure.

(38) (from 34,37) $\neg W'(A, w_3)$
$/\{\langle 0 \stackrel{K_A}{\rightarrow} Sk_2 \stackrel{K_B}{\rightarrow} w_3 \rangle\}$
(39) (from 35,38) $W'(B, w_2)/\{\langle 0 \stackrel{K_A}{\rightarrow} Sk_2 \stackrel{K_B}{\rightarrow} w_2 \rangle\}$
(40) (from 36,39) $\square$

In step (38) the primary deduction mechanism (resolution) has returned the substitution $\{SK_2/w_1\}$. Thus we must *UNIFY* the sequences $\langle 0 \stackrel{K_A}{\rightarrow} w_1 \rangle$ and $\langle 0 \stackrel{K_A}{\rightarrow} Sk_2 \rangle$. This succeeds with the substitution $\{SK_2/w_1\}$. The variable w_2 in (38) has been renamed to w_3 to avoid a clash in the next step. In step (39), the outer deduction results in the substitution $\{w_2/w_3\}$. We must then *UNIFY* $\langle 0 \stackrel{K_A}{\rightarrow} SK_2 \stackrel{K_B}{\rightarrow} w_3 \rangle$ and $\langle 0 \stackrel{K_A}{\rightarrow} w_7 \stackrel{K_B}{\rightarrow} w_2 \rangle$. This results in the substitution $\{SK_2/w_7, w_2/w_3\}$. In the last step, the empty clause is produced with the substitution $\{SK(w_1)/w_2\}$. This requires *UNIFY* to operate on $\langle 0 \stackrel{K_A}{\rightarrow} SK_2 \stackrel{K_B}{\rightarrow} w_2 \rangle$ and $\langle 0 \stackrel{K_A}{\rightarrow} w_1 \stackrel{K_B}{\rightarrow} SK(w_1) \rangle$. The unification succeeds with the substitution $\{SK_2/w_1, SK(w_1)/w_2\}$. Note that clauses (35) and (37) can not resolve because the sequences $\langle 0 \stackrel{K_A}{\rightarrow} w_1 \stackrel{K_B}{\rightarrow} w_2 \rangle$ and $\langle 0 \stackrel{K_A}{\rightarrow} Sk_2 \rangle$ do not unify; they are of different lengths. Notice that the problems of increasingly large constraints and of work performed at earlier steps needing to be redone have been eliminated.

Since, in the case of the modal logic D, the procedure *UNIFY* is ordinary term unification, sequences which are not the same length cannot be made the same by any substitution and therefore cannot unify. Now consider the least Herbrand model of a Σ from the reified modal logics for a logic with additional conditions on the accessibility relation (e.g. T, S_4. and S_5). Additional connections are created in the graph representation of the least Herbrand model of Σ by the axioms of reflexivity, symmetry, and transitivity. It is no longer the case that there is a unique path to every term. But in all these cases, the set of terms and paths to those terms may be represented as a tree which is then closed under the appropriate axioms. This closure is captured by performing unification in the presence of the additional axioms. It is no longer the case that the sequences must have the same length in order to unify. For example, in the presence of reflexivity, the clauses (35) and (37) can resolve because the sequences $\langle 0 \stackrel{K_A}{\rightarrow} w_1 \stackrel{K_B}{\rightarrow} w_2 \rangle$ and $\langle 0 \stackrel{K_A}{\rightarrow} Sk_2 \rangle$ do unify with substitution $\{Sk_2/w_1, Sk_2/w_2\}$.

The procedure *UNIFY* from Figure 3 is unification in the presence of the appropriate axioms. With the additional axioms, unification is no longer guaranteed to find a single maximally general unifier. It may produce multiple maximally general unifiers—but only a finite number. Consider the unification of sequences $\langle 0 \stackrel{K_A}{\rightarrow} w_1 \stackrel{K_A}{\rightarrow} Sk \rangle$ and $\langle 0 \stackrel{K_A}{\rightarrow} w_2 \rangle$. Unification in the presence of reflexivity yields two maximally general unifiers—$\{0/w_1, Sk/w_2\}$ and $\{Sk/w_1, Sk/w_2\}$. Now that *UNIFY* returns a set of substitutions, the sequence unification procedure illustrated in Figure 3 must be modified to return a set of pairs, with each pair consisting of a substitution and a set of sequences. The production of multiple maximally general unifiers is a result of this particular representation for the constraints and is not a characteristic of the general method. In the

have come from two sources—Skolemization and the accessibility axioms. All of the accessibility axioms are represented implicitly in the sequence unification procedure. The remaining definite clauses that make up Σ are represented as sequences. For example, the clause $K(0, w_1) \rightarrow K(w_1, sk(w_1))$ is represented as $\langle 0 \stackrel{K}{\rightarrow} w_1 \stackrel{K}{\rightarrow} Sk(w_1)\rangle$. Now Σ is a set of sequences and each of the constraints attached to the clauses is a set of sequences.

It is an important property of the sequence representation of the constraints and Σ that for any symbol occurring in the sequences resulting from the translation of a set of modal sentences, there is only one distinct sequence terminating in that symbol. This is guaranteed by the following theorem.

Theorem 8 *The constraint theory Σ and all constraints on the A-clauses in a reified modal logic when represented as a set of sequences form a tree with uniquely labeled nodes— i.e. there is only one path from the root to any node in the tree.*

Therefore, for any symbol occurring in a set of sequences (representing either a constraint or Σ in a reified modal logic), there is, in the set of prefixes of the sequences in the set, a unique sequence terminating in that symbol.

Let us consider how the reified modal logic representation of the wise-man problem can be represented with the sequence notation. As there are two accessibility relations, two distinct labels (K_A and K_B) are needed for the edges of the sequences. Clauses (15) – (18) can be written with the constraints in sequence form as follows:

(34) (axiom) $W'(A, w_1) \vee \neg W'(A, w_2)$
$/\{\langle 0 \stackrel{K_A}{\rightarrow} w_1 \stackrel{K_B}{\rightarrow} w_2\rangle\}$
(35) (axiom) $W'(A, w_2) \vee W'(B, w_2)$
$/\{\langle 0 \stackrel{K_A}{\rightarrow} w_1 \stackrel{K_B}{\rightarrow} w_2\rangle\}$
(36) (axiom) $\neg W'(B, Sk(w_1))/\emptyset$
(37) (axiom) $\neg W'(A, Sk_2))/\emptyset$

The constraint theory Σ_{2_w} is represented as a set of two sequences— $\{\langle 0 \stackrel{K_A}{\rightarrow} Sk_2\rangle, \langle 0 \stackrel{K_A}{\rightarrow} w_1 \stackrel{K_B}{\rightarrow} Sk(w_1)\rangle\}$.

Using this representation, the sequence unification algorithm, shown in Figure 3, can be used to determine the Σ-solvability of the constraints. The input to the algorithm consists of L_1 and L_2 (the two literals of opposite polarity from a step of potential constrained resolution of two clauses), the constraints C_1 and C_2 (both sets of sequences) from the clauses and a constraint theory Σ (a set of sequences). The output of the algorithm is either $FAIL$, indicating that L_1 and L_2 do not share any Σ-ground instances and therefore that the new clause cannot

Input: two expressions L_1 and L_2, and three sets of sequences C_1, C_2, and Σ. L_1 and C_1 share no variables with L_2 and C_2.
Output: $FAIL$ or a set of sequences C and a substitution λ.

If L_1 and L_2 are not unifiable
  halt and report FAIL
Else
  $\theta :=$ mgu(L_1,L_2);
$\lambda := \{\}$;
$C := C_1 \cup C_2$;

For each variable w_i such that $w_i\theta \neq w_i$, do the following:
  $s :=$ the prefix ending in w_i of a sequence in C;
  If $w_i\theta$ is a variable then
    $s' :=$ the prefix of a sequence in C $w_i\theta$
  Else ($w_i\theta$ is of the form $f(t_1, \ldots t_n)$
      for some $n \geq 0$)
    $s' :=$ the prefix of a sequence in Σ such that
    it ends with $f(t_1', \ldots t_n')$ for some $t_1' \ldots t_n'$;
  If $UNIFY(s, s')$ succeeds with substitution γ
  then
    $\lambda :=$ the composition of λ and γ
    $C := C\gamma$
  Else
    halt and report FAIL;
If the procedure finishes without reporting FAIL, C is returned by the procedure along with λ.

Figure 3: Constraint Solving Procedure

be deduced, or a new constraint C (equivalent to $(C_1 \wedge C_2)\theta$) in the form of a set of sequences and, a substitution λ. If the procedure does not result in $FAIL$, the substitution λ is applied to the body of the new clause, and the constraint of the new clause is C. The procedure $UNIFY$ will vary according to what modal logic is being used. For D, $UNIFY$ is ordinary term unification. The case of other logics will be discussed later.

In establishing the correctness of this procedure for testing Σ-solvability, we consider the least Herbrand model of a Σ in a reified modal logic for the modal logic D. In this model the connectivity relations (e.g. K_A, K_B) denote tuples of ground terms from the Herbrand Universe of the clauses in Σ. The connectivity relations together form a graph in which the edges are labeled with the connectivity relations and the nodes are the ground terms.

Theorem 9 *The graph of the connectivity relations from the least Herbrand model of a Σ for the modal logic D is a tree with the root being the term 0.*

need to be redone later, but is rather stored in the representation of the constraints. Observe that the work done in the example above on testing for Σ-solvability at each step is redone again and again at later steps. If the testing for Σ-solvability is done by some sort of deduction (e.g. SLD resolution), the fact that the only term which can instantiate w_2 is $SK(Sk_2)$ (in step (21) above) will be deduced again and again—whenever the test for Σ-solvability is performed. The specialized constraint processing method, to be presented in the next section, is incremental.

Before continuing to the next section, we give an illustration (based on an example in [Moore, 1980]) of the operation of the rule of constrained paramodulation.

(23) $\Box num(bill) = num(mary)$
(24) $\exists x \Box x = num(bill)$

An agent (23) knows that Bill's phone number is the same as Mary's phone number. Additionally, (24) the agent knows Bill's phone number. The hypothesis to be proven is (25) that the agent knows Mary's phone number.

(25) $\exists x \Box x = num(mary)$

The translation into the reified modal logic is as follows, where (28) is the translation of the negation of sentence (25):

(26) $\forall w_1{:}K(0, w_1)$
$num(bill, w_1) = num(mary, w_1)$
(27) $\exists x \, \forall w_1{:}K(0, w_1) \; x = num(bill, w_1)$
(28) $\neg \exists x \, \forall w_1{:}K(0, w_1) \; x = num(mary, w_1)$

Putting (26)—(28) in clausal form results in the following:

(29) $num(bill, w_1) = num(mary, w_1)/K(0, w_1)$
(30) $f = num(bill, w_1)/K(0, w_1)$
(31) $\neg(x = num(mary, sk(x)))/\emptyset$

The constraint theory for this problem contains the clause $K(0, sk(x))$ along with the accessibility axioms for the modal logic being used. In this particular example, no accessibility axioms are needed to obtain a proof.

A proof that (29)—(31) are unsatisfiable is obtained by first paramodulating clause (29) into clause (30) to obtain (32).

(32) $f = num(mary, w_1)/K(0, w_1)$

Next, the empty clause (33) is obtained by resolving clauses (32) and (31).

(33) $\Box/K(0, sk(x))$

Compare this with the situation where (23) is replaced by

(23′) $num(bill) = num(mary)$.

The clausal reified modal logic translation of (23′) is

(29′) $num(bill, 0) = num(mary, 0)/\emptyset$.

Now (29′) can be paramodulated into (30) to yield

(32′) $f = num(mary, 0)/K(0, 0)$

provided $K(0, 0)$ is Σ-solvable. Since it is not, the proof fails. The step would, however, succeed in a reflexive logic because it would then be the case that $\Sigma \models K(0, 0)$. But the proof of the unsatisfiability of (29′), (30) and (31) still fails because (32′) and (31) can not resolve.

4 Testing for Σ-Solvability

This section presents an alternative representation for the constraints of a reified modal logic and a procedure for testing their Σ-solvability. The procedures are specialized unification algorithms that operate on constraints represented as sequences of symbols that range over or denote worlds. This method has been utilized in a number of the related methods [Ohlbach, 1988, Wallen, 1990, Jackson and Reichgelt, 1989, Auffray and Enjalbert, 1989] mentioned earlier; our contribution is to show that these methods can be viewed as algorithms for testing Σ-solvability of constraints. The method is applicable to modal logics for which the translation procedure adds some subset of the reflexive, symmetric and transitive axioms to the constraint theory (along with the serial axiom). It is not applicable to logics that call for the Euclidian axiom to be added to the constraint theory.[2]

Let us define a *sequence* as a unary tree with labeled edges and vertices. The root label is always the constant 0, the other vertices are labeled with terms and the edge labels are the K predicate symbols. The constraints that occur in the reified modal logic can be represented as sets of sequences. For example, $K(0, Sk_1) \wedge K(Sk_1, w_1) \wedge K(w_1, f(w_1)) \wedge K(f(w_1), w_2)$ is represented as $\{(0 \xrightarrow{K} Sk_1 \xrightarrow{K} w_1 \xrightarrow{K} f(w_1) \xrightarrow{K} w_2)\}$.

A constraint theory can also be represented as sequences. Recall that the definite clauses within Σ

[2] Of the 9 distinct modal logics to which the general framework is applicable, the special procedure for testing the Σ-solvability of constraints is not applicable to $KD5$ and and $KD45$. It is applicable to $S5$ because, although $S5$ is a Euclidian logic, the conditions on the accessibility relation can be axiomatized without the Euclidian axiom.

where θ is a substitution used in the deduction. The corresponding constraint logic rule is

From ϕ_1/C_1 and ϕ_2/C_2 derive $(\phi_3/C_1 \wedge C_2)\theta$, provided that $(C_1 \wedge C_2)\theta$ is Σ-solvable.

The provision that $(C_1 \wedge C_2)\theta$ is Σ-solvable ensures that the derived formula has at least one ground instance. Without this provision, the inference rule could be used to produce a proof that is not schematic for any ground proof, thereby compromising soundness. The only place in a substitutional reasoner where Σ is used is in testing this provision.

The above schema for constraint deduction can be instantiated to yield the rules of constraint resolution and constraint paramodulation. Simply consider $\phi_3\theta$ to be a resolvent (paramodulant) of the clauses ϕ_1 and ϕ_2, where θ is the most general unifier used in performing the resolution (paramodulation).

The justification for handling constraints in this manner parallels that for using unification to handle variables without constraints: Every ground clause that can be obtained by constraint resolution (paramodulation) from a Σ-ground instance of ϕ_1/C_1 and a Σ-ground instance of ϕ_2/C_2 is a Σ-ground instance of a constraint resolvent (paramodulant) of ϕ_1/C_1 and ϕ_2/C_2. This property is asserted by the following lifting theorem:

Theorem 6 (Lifting Theorem) *Let Σ be a constraint theory, let α be a set of constrained clauses, and let ϕ be a ground clause. If ϕ is derivable from $\alpha_{\Sigma gr}$ by constraint resolution and constraint paramodulation, then some constrained clause that has ϕ as a Σ-ground instance is derivable from α by constraint resolution and constraint paramodulation.*

The proof of a lifting theorem for a constraint deduction system can be systematically produced from a proof of the corresponding theorem for ordinary deduction. The basis for this is that lifting theorems for ordinary deduction systems hold because of certain properties of instantiation, and it can be shown that instantiation for constrained formulas has all these crucial properties.

Putting this all together yields a proof of completeness theorem for constraint resolution and constraint paramodulation.

Theorem 7 (Completeness Theorem) *Let α be a set of constrained clauses and Σ be a constraint theory with a least Herbrand model. If $\Sigma \cup \alpha$ is unsatisfiable then constraint resolution and constraint paramodulation can derive the empty clause from α using the constraint theory Σ.*

If $\Sigma \cup \alpha$ is unsatisfiable, then, by the Constraint Herbrand Theorem, so is $\alpha_{\Sigma gr}$. When operating on clauses that contain no constraints, constraint resolution (paramodulation) is identical to ordinary resolution (paramodulation), and therefore constraint resolution (paramodulation), like ordinary resolution (paramodulation), can derive the empty clause from $\alpha_{\Sigma gr}$. Finally, the Lifting Theorem enables us to conclude that constraint resolution and constraint paramodulation can derive the empty clause from α using the constraint theory Σ.

The following refutation of clauses (15)–(18) using the constraint theory Σ_{2wm} illustrates the use of constraint resolution.

(19) $\neg White'(A', w_2)/K_A(0, Sk_2) \wedge K_B(Sk_2, w_2)$ (from 15,18)

(20) $\neg White'(A', w_2)/K_B(Sk_2, w_2)$ (simplification of 19)

(21) $White'(B', w_2)/K_B(Sk_2, w_2) \wedge K_A(0, w_1) \wedge K_B(w_1, w_2)$ (from 16,20)

(22) $\Box/K_B(Sk_2, Sk(w_4)) \wedge K_A(0, w_1) \wedge K_B(w_1, Sk(w_4)) \wedge K_A(0, w_4)$ (from 21,17)

At each step it is necessary to verify that the new constraint is Σ-solvable, thereby eliminating some potential resolvents. For example, to resolve clause (18) with the first literal of clause (16) it is necessary to test the constraint $K_A(0, w_1) \wedge K_B(w_1, Sk_2)$ for Σ-solvability. This constraint is not Σ-solvable since $\Sigma_{2wm} \not\models \exists K_A(0, w_1) \wedge K_B(w_1, Sk_2)$. But it would be Σ-solvable if Σ contained the reflexivity axiom—e.g. if we were using the modal logic T.

The efficiency of a constraint deduction system depends critically upon the capability of the procedure to simplify constraints by replacing them with equivalent, but shorter constraints. This derivation uses one obvious constraint simplification. At step (20) a ground atom is deleted because it is Σ-solvable. Ideally, we would like to reduce the constraint to an equivalent maximally reduced [Buntine, 1988] or smallest constraint. In (21), the constraint $K_B(Sk_2, w_2) \wedge K_A(0, w_1) \wedge K_B(w_1, w_2)$ could be simplified to $K_B(Sk_2, w_2)$, but this is a non-trivial simplification. It requires realising that for any ground terms t_1 and t' such that $\Sigma_{2wm} \models K_B(t_1, t')$, there will not be another t_2 such that $\Sigma_{2wm} \models K_B(t_2, t')$. In other words, if one interprets the minimal Herbrand model of Σ_{2wm} as a system of interconnected worlds, there can only be one distinct path to any world—the set of interconnected worlds form a tree.

This property is exploited by the specialized procedure, to be presented in the next section, for simplifying and testing the Σ-solvability of constraints that arise in a reified modal logic. The special procedure should also be incremental in that work done in testing for Σ-solvability at one point, does not

other times additional mechanisms must be incorporated. In either case, the idea is that deductions on quantified sentences are themselves schematic for deductions on ground sentences. A schematic deduction is said to *lift* each deduction that is an instance of it.

Given a deduction system based on unification we would like to show that it does indeed treat quantified sentences as schemas. This usually takes the form of a *lifting theorem* for the deduction system stating that every deduction that can be made from the ground instances of a set of sentences can be made schematically from the sentences themselves.

As an example to illustrate these ideas, consider resolving $P(x,a) \vee Q(x,y)$ with $\neg P(a,z)$. Herbrand's Theorem tells us to consider each of these clauses as standing for the set of all its ground instances. Observe that every ground clause that can be obtained by resolving a ground instance of the first clause with a ground instance of the second is an instance of $Q(a,y)$. Through the use of unification, resolution for non-ground clauses is designed so that it produces $Q(a,y)$ when it resolves the two non-ground clauses together. The Lifting Theorem for Resolution states that this relationship is always obtained.

A substitutional system for constraint logic treats quantified sentences as sentence schemas in much the same way that an ordinary deduction system does. The only difference is that a sentence with constraints does not stand for the set of all its ground instances, but only for those obtained by replacing variables with terms that respect the constraints attached to the variables. Whether or not the constraints are respected depends upon the constraint theory Σ. The ground instances of a constrained formula are therefore referred to as Σ-*ground instances* and defined as such:

Definition 4 *Let Σ be a constraint theory, e/C be a constrained expression, and θ be a substitution such that $e\theta$ is ground. Then $e\theta$ is a* Σ-ground instance *of e/C if $\Sigma \models \exists\, C\theta$.*[1] *The set of all Σ-ground instances of e is denoted by $e_{\Sigma gr}$.*

Observe that by this definition a ground instance of a constrained expression has no variables and *no constraints.* Also observe that some constrained expressions, such as $P(x)/Q(x) \wedge \neg Q(x)$, have no Σ-ground instances. We say that a constraint C is Σ-*solvable* if $\Sigma \models \exists C$. Hence, a constrained expression, e/C, has a Σ-ground instance if, and only if, C is Σ-solvable.

This treatment of restricted quantifiers is justified by the *Constraint Herbrand Theorem*, which follows this paragraph. This theorem states that, under certain circumstances, a constraint theory Σ and a set α of constrained sentences are jointly satisfiable if, and only if, the set of all Σ-ground instances of α is satisfiable. In saying that α and Σ can be replaced by the Σ-ground instances of α, the Constraint Herbrand Theorem justifies using Σ only in instantiating formulas. Notice that Σ must be used in generating the correct instances of α, but once they are obtained Σ is irrelevant because ground instances have no constraints.

Theorem 5 (Constraint Herbrand Theorem) *Let α be a set of constrained Skolem normal form sentences with only positive constraints and let Σ be a constraint theory that has a least Herbrand model. Then $\alpha \cup \Sigma$ is unsatisfiable if, and only if $\alpha_{\Sigma gr}$ is unsatisfiable.*

The rules of resolution and paramodulation operate on clauses. Now, since the convention is to drop quantifiers when converting to clause form,

$$\forall w_1{:}K(0,w_1)\ \ldots \forall w_n{:}K(w_{n-1},w_n)\ \varphi$$

where φ is a disjunction of literals can be written as

$$\varphi / K(0,w_1) \wedge \ldots \wedge K(w_{n-1},w_n).$$

With this convention, after all existential quantifiers have been eliminated, universal quantifiers may be dropped, the remaining formula can be put in conjunctive normal form and separated into clause form. Each separate clause will have a (possibly empty) constraint expression which is a conjunction of K-atoms expressing the constraints on the variables in that clause.

Translating (11) – (14) into constrained clauses yields (15) – (18) respectively.

(15) $White'(A', w_1) \vee \neg White'(A', w_2)$
$\quad /K_A(0,w_1) \wedge K_B(w_1,w_2)$
(16) $White'(A', w_2) \vee White'(B', w_2)$
$\quad /K_A(0,w_1) \wedge K_B(w_1,w_2)$
(17) $\neg White'(B', Sk(w_1))/K_A(0,w_1)$
(18) $\neg White'(A', Sk_2))$

Σ_{2wm} remains as given earlier.

A deduction system for constraint logic can be produced systematically from an ordinary first-order deduction system by integrating a special-purpose constraint processing mechanism into it. Suppose that the ordinary deduction system has an inference rule of the form

From ϕ_1 and ϕ_2 derive $\phi_3\theta$,

[1] Here, and in general, an expression of the form $\exists\phi$ denotes the existential closure of ϕ—that is, the formula $\exists x_1 \cdots x_n\ \phi$ where $x_1, \ldots, x_n$ are the freely-occurring variables of ϕ.

(5) $\forall w_1{:}K_A(0,w_1)\ (\neg White'(A',w_1)$
$\rightarrow \forall w_2{:}K_B(w_1,w_2)\ (\neg White'(A',w_2)))$
(6) $\forall w_1{:}K_A(0,w_1)\ \forall w_2{:}K_B(w_1,w_2)$
$(White'(A',w_2) \vee White'(B',w_2))$
(7) $\forall w_1{:}K_A(0,w_1)\ \exists w_2{:}K_B(w_1,w_2)$
$\neg White'(B',w_2)$
(8) $\exists w_1{:}K_A(0,w_1)\ \neg White'(A',w_1))$
(negated hypothesis)

The axioms

(9) $\forall w_1\ K_A(w_1, SK'(w_1))$ (serial axiom for K_A)
(10) $\forall w_1\ K_B(w_1, SK''(w_1))$ (serial axiom for K_B)

are added to the constraint theory for this problem, which we call Σ_{2wm}. For simplicity, these have been entered in Skolem Normal form. Translating (5) – (8) into constrained clauses yields (11) – (14) respectively.

(11) $\forall w_1{:}K_A(0,w_1)\ \neg White'(A',w_1)$
$\rightarrow \forall w_2{:}K_B(w_1,w_2)\ (\neg White'(A',w_2)))$
(12) $\forall w_1{:}K_A(0,w_1)\ \forall w_2{:}K_B(w_1,w_2)$
$(White'(A',w_2) \vee White'(B',w_2))$
(13) $\forall w_1{:}K_A(0,w_1)\ \neg White'(B', Sk(w_1))$
(14) $\neg White'(A', Sk_2)$ (negated hypothesis)

Σ_{2wm} contains the following sentences.

a. $\forall w_1\ K_A(0,w_1) \rightarrow K_B(w_1, Sk(w_1))$
(from Skolemization of 7)
b. $K_A(0, Sk_2)$ (from Skolemization of 8)
c. $\forall w_1\ K_A(w_1, SK'(w_1))$ (from 9)
d. $\forall w_1\ K_B(w_1, SK'(w_1))$ (from 10)

To summarize, the transformations given in this section translate a set of modal sentences into reified modal logic. The reified modal logic that results takes the form of a constraint logic—that is, it consists of a set of constrained formulas and a constraint theory. The next section presents a class of deduction systems that operate on an any constraint logic, regardless of whether the constrained formulas form a reified modal logic.

3 Constraint Deduction

This section presents a general framework for constructing and analyzing a particular class of hybrid deduction systems, called substitutional systems, for constraint logic. This substitutional framework provides a general method for systematically transforming an automated deduction system for FOPC into one for constraint logic, and for transforming the completeness proof of the original deduction system into one for the resulting deduction system. These transformations can be applied to any first-order deduction system that treats variables schematically through the use of unification.

This section presents the major ideas underlying the substitutional framework and states, but does not prove, the fundamental theorems. The theorems presented here generalize, in two ways, those that Frisch [1991] developed for a sorted logic called SFOPC. The atomic constraints in SFOPC are composed of monadic predicates, called sort symbols, applied to variables. So the first generalization here is that atomic constraints may be any atomic formula whose predicate symbol is a constraint symbol. The second generalization is that the language contains the equality symbol, which is precluded from the previous work. The equality symbol is interpreted as a logical symbol denoting identity. It is not a constraint symbol, so it may occur in the body of an A-formula, but it may not appear in any constraints on the formulas of the sort theory.

Space precludes in depth discussion of how the framework can be used to construct a wide range of deductive rules for constraint logic, so this section concentrates on ordinary resolution and paramodulation. To further simplify the presentation we make the assumption, which is not necessary, that all constraints on A-formulas are positive—that is, if they were transformed into conjunctive or disjunctive normal form they would contain no negation signs. As all constraints in reified modal logic are positive, this assumption poses no problems for our present use of constraint deduction. We also assume that all sentences are in Skolem Normal Form. Thus, the words "sentence" and "theory" only refer to sentences and theories in Skolem Normal Form.

An understanding of the role of unification in ordinary deduction forms the basis upon which the substitutional framework is built. Since the introduction of resolution in 1965, nearly every automated deduction system has handled universally quantified variables by using a unification algorithm. By paralleling this general approach to deduction with ordinary variables, the substitutional framework for deduction with constrained variables achieves the same degree of generality.

The gist of this approach is that a quantified sentence is treated as a schema standing for the set of its ground instances. The justification for this is found in Herbrand's Theorem, which states that a set of quantified sentences is satisfiable if, and only if, the set containing every ground instance of every one of the sentences is satisfiable. Thus, a formula such as $P(x) \vee Q(x)$ can be read either as $\forall x\ P(x) \vee Q(x)$ or as a schema standing for the set of all its ground instances.

A deduction system that operates on ground sentences can be made to operate on quantified sentences by replacing tests for equality between expressions with tests for unifiability between them. Sometimes this is all that needs to be done, though

The correctness of the translation function $\mathbf{T_0}$ is established by the following theorem.

Theorem 1 *Let $\mathcal{L}_m$ be a modal logic and let $\mathcal{A}$ be the axioms that define the conditions on the accessibility relation of the possible world semantics for $\mathcal{L}_m$. For any set of modal sentences α of $\mathcal{L}_m$, α is satisfiable iff $\mathbf{T_0}(\alpha) \cup \mathcal{A}$ is satisfiable.*

Now that the modal sentences have been translated to constrained sentences, we turn to the second topic of the section—the translation of constrained sentences to constrained clauses while preserving satisfiability. The first step is to put the constrained sentences in prenex form. The following theorem establishes that any constrained formula may be rewritten to a logically equivalent formula in prenex form by using six equivalence schemata (derived from [Frisch, 1991]).

Theorem 2 (Prenex Transform Theorem)
If φ and ψ are formulas and ψ has no free occurrences of the variable y and t is an arbitrary term, then:

$$\neg\, \exists y{:}K(t,y)\ \varphi \equiv \forall y{:}K(t,y)\ \neg\, \varphi$$
$$\neg \forall y{:}K(t,y)\ \varphi \equiv \exists y{:}K(t,y)\ \neg\, \varphi$$
$$(\exists y{:}K(t,y)\ \varphi)\ \wedge \psi \equiv \exists y{:}K(t,y)\ (\varphi \wedge \psi)$$
$$(\forall y{:}K(t,y)\ \varphi)\ \vee \psi) \equiv \forall y{:}K(t,y)\ (\varphi \vee \psi)$$

Furthermore, if Σ contains the seriality axiom—$\forall w_1\, \exists w_2\ K(w_1, w_2)$—then:

$$((\exists y{:}K(t,y)\ \varphi)\ \vee \psi) \cup \Sigma \equiv \exists y{:}K(t,y)\ (\varphi \vee \psi) \cup \Sigma$$
$$((\forall y{:}K(t,y)\ \varphi)\ \wedge \psi) \cup \Sigma \equiv \forall y{:}K(t,y)\ (\varphi \wedge \psi) \cup \Sigma$$

Using these equivalences, the statements produced by the translation function can be put in prenex normal form by moving all quantifiers (constrained and otherwise) to the front.

The next step is the elimination of constrained existential quantifiers through Skolemization. Skolemisation of constrained variables occurs as with ordinary variables, except that information must be added to Σ. For example, consider the constrained variable y in $\exists y{:}K(t,y)\ P(y)$. The variable y is replaced by some Skolem function, say SK, yielding $P(SK)$. Additionally, $K(t, SK)$ must be added to Σ. All quantifiers in the translated sentences will be of the form $Q_i\, x_i{:}C_i$ where Q_i is either an existential or universal quantifier and C_i is a possibly empty constraint. If the constraint is empty, the variable x_i is an ordinary unconstrained variable. Otherwise, the constraint is $K(t_{i-1}, x_i)$. The general procedure for the Skolemization of constrained variables is given in Figure 2 and its correctness is established by the following theorem.

Theorem 3 (Skolem Transform Theorem)
$\alpha \cup \Sigma$, a set of prenex form constrained sentences, is satisfiable iff its Skolem Normal Transform is.

Input: A set α of prenex form sentences, and a constraint theory Σ.

While α contains a sentence ϕ that contains an existential quantifier do:
Observe that ϕ is of the form

$$\forall x_1{:}C_1 \ldots \forall x_{n-1}{:}C_{n-1}\ \exists x_n{:}C_n\ M$$

where $n \geq 0$, M is a prenex-form formula, and each C_i is of the form $K(t_{i-1}, x_i)$ (in which case x_i is a constrained variable) or is empty (in which case x_i is an ordinary unconstrained variable).

Choose f, some n-ary function symbol that does not occur in α or Σ and let θ be $\{f(x_1 \ldots, x_{n-1})/x_n\}$. Replace ϕ in α by the sentence

$$\forall x_1{:}C_1 \ldots \forall x_{n-1}{:}C_{n-1}\ (M\theta)$$

and if C_n is not empty, add

$$\forall x_1 \ldots x_{n-1} C_1 \wedge \ldots \wedge C_{n-1} \rightarrow (C_n\theta)$$

to Σ.

Figure 2: Constrained Skolemization Procedure

Note also the following property of this procedure.

Proposition 1 *When the Skolemization procedure is applied to the constrained sentences translated from modal sentences, all of the clauses added to the constraint theory Σ are definite clauses.*

The definite clauses created by the constrained Skolemization procedure are added to the constraint theory Σ along with the axioms that define the restrictions on the accessibility relation in the possible world semantics for the logic under consideration. Since the axioms are all definite clauses, Σ contains only definite clauses. Thus the final result is a set of constrained clauses as well as the definite clause theory Σ.

As an illustration of the entire translation, consider the wise-man puzzle given earlier. The modal system to be used is D—the minimal serial modal logic. In this case, we have to utilise distinct K predicates (K_A and K_B) to represent the accessibility relations corresponding to the modal operators $\Box_A$ and $\Box_B$. We treat the names A and B as rigid designators and therefore do not translate them as denoting relative to a possible world—i.e. with an additional argument position. Applying the translation to (1) – (4) yields:

(1) $\Box_A(\neg White(A) \rightarrow \Box_B(\neg White(A)))$
(2) $\Box_A(\Box_B(White(A) \vee White(B)))$
(3) $\Box_A(\neg\Box_B(White(B)))$
(4) $\neg\Box_A(White(A))$

Figure 1: Wise Man Puzzle

cessibility relation for each of the modal operators. Figure 1 contains the representation of this puzzle in modal logic. The puzzle is formulated for a refutational proof procedure with the last line being the negation of the hypothesis to be proven. We will see that the minimal serial modal logic D suffices to obtain a solution to this puzzle.

Section 2 covers the procedure for translating modal sentences to constrained sentences in prenex normal form. In Section 3, the substitutional framework for the construction of systems of hybrid reasoning is used to modify ordinary resolution to obtain a form of resolution for constrained clauses. Section 4 covers a specialized algorithm for modal constraint processing. Sections 2 through 4 assume the modal logic is a constant domain modal logic. Section 5 briefly discusses how the framework can handle varying domain modal logics. In the conclusion, Section 6, future research is discussed.

2 Translation to Constrained Clauses

The first part of this section describes a procedure for translating sentences in a constant domain modal logic into a non-modal language with constrained quantifiers. We call the resulting set of sentences a *reified modal logic*. The second part of the section describes a procedure for converting these sentences into Skolem normal form. As both of the translations preserve satisfiability, the initial set of modal sentences is satisfiable if and only if the resulting set of Skolem normal form sentences of reified modal logic is satisfiable.

The translation from modal logic to reified modal logic introduces the predicate K as a constraint on variables that range over possible worlds. The constraints on sentences of the reified modal logic represent the connections between possible worlds in the Kripke semantics for the modal logic. Sentences in modal logic are evaluated in a *model structure* [Kripke, 1963], which is a system of interconnected elements called possible worlds with one of the possible worlds designated as the actual world. If a and b are two worlds, $K(a,b)$ means that b is accessible from a. Then, every proposition true in b is possible in a. A proposition is necessary in world a, if it is true in every world accessible from a.

For example, $\Box P$ in a propositional modal logic is translated into the constrained sentence $\forall x{:}K(0,x)\ P(x)$, which is an abbreviation for $\forall x\ K(0,x) \rightarrow P(x)$. This sentence can be read as saying for all worlds x, if x is accessible from world 0, which denotes the actual world, then P is true in x. Similarly, $\Diamond P$ is translated into the constrained sentence $\exists x{:}K(0,x)\ P(x)$, which is an abbreviation for $\exists x\ K(0,x) \wedge P(x)$. This sentence can be read as saying that there is some world x that is connected to the actual world 0 and P is true in x.

The translation function $\mathbf{T_0}$ maps sets of modal sentences to sets of constrained sentences while preserving satisfiability. The translation function at an arbitrary step is represented by $\mathbf{T_i}$ where $i \geq 0$. The integer i is initially set to 0 and is incremented at various points in the translation process The translation function is recursively defined as follows:

- $\mathbf{T_0}(S) = \{\mathbf{T_0}(s) | s \in S\}$, where S is a set of sentences.
- $\mathbf{T_i}(x) = x'$, where x is a variable.
- $\mathbf{T_i}(f(t_1,\ldots,t_n)) = f'(\mathbf{T_i}(t_1),\ldots,\mathbf{T_i}(t_n),w_i)$ where $f(t_1,\ldots,t_n)$ is a term
- $\mathbf{T_i}(a) = a'(w_i)$, where a is a constant.
- $\mathbf{T_i}(P(t_1,\ldots,t_n)) = P'(\mathbf{T_i}(t_1),\ldots,\mathbf{T_i}(t_n),w_i)$ where P is a predicate.
- $\mathbf{T_i}(\varphi \wedge \psi) = \mathbf{T_i}(\varphi) \wedge \mathbf{T_i}(\psi)$
- $\mathbf{T_i}(\neg\varphi) = \neg\mathbf{T_i}(\varphi)$
- $\mathbf{T_i}(\varphi \vee \psi) = \mathbf{T_i}(\varphi) \vee \mathbf{T_i}(\psi)$
- $\mathbf{T_i}(\forall x\varphi) = \forall x'\ \mathbf{T_i}(\varphi)$
- $\mathbf{T_i}(\exists x\varphi) = \exists x'\ \mathbf{T_i}(\varphi)$
- $\mathbf{T_i}(\Box\varphi) = \forall w_{i+1}{:}K(w_i,w_{i+1})\ \mathbf{T_{i+1}}(\varphi)$
- $\mathbf{T_i}(\Diamond\varphi) = \exists w_{i+1}{:}K(w_i,w_{i+1})\ \mathbf{T_{i+1}}(\varphi)$

In the sentences that result from the translation, w_0 is a constant symbol which denotes the actual world, but it is replaced by 0 to distinguish it clearly from the other w_i, each of which is a variable. The translation function is similar to those proposed by Moore[1980] and Morgan [1976].

The constraint theory Σ must contain the axioms that define the conditions on the accessibility relation for the modal logic being used. For each of the logics under consideration here, a subset of the following axioms is needed:

Serial: $\forall w_1\ \exists w_2\ K(w_1,w_2)$
Reflexive: $\forall w_1\ K(w_1,w_1)$
Euclidean: $\forall w_1,w_2,w_3\ K(w_1,w_2) \wedge K(w_1,w_3) \rightarrow K(w_2,w_3)$
Symmetric: $\forall w_1,w_2\ K(w_1,w_2) \rightarrow K(w_2,w_1)$
Transitive: $\forall w_1,w_2,w_3\ K(w_1,w_2) \wedge K(w_2,w_3) \rightarrow K(w_1,w_3)$.

of an entity in a world, are also needed. Deduction on the resulting sentences is performed by a non-modal deduction system that is extended with special purpose built-in reasoners that handle the K- and E-atoms.

Now let us look at the method more closely. A modal deduction system produced by our method operates on non-modal sentences of a language that we call constraint logic. The sentences of constraint logic are like sentences of ordinary first-order logic except that tuples of variables may be associated with constraints that restrict the domain over which the variables range. Sentences of constraint logic are syntactic variants of ordinary first order sentences since the constraints can be translated out. The name constraint logic is chosen because the logic can be seen as a generalization of Horn-clause constraint logic programming [Jaffar and Lassez, 1987, Van Hentenryk, 1989] to full first-order logic.

The constraint logic sentences are obtained by translating the modal sentences on which one wishes to perform deduction. Based on the possible-worlds semantics of modal logic, a modal sentence can be viewed as implicit discourse about possible worlds and the accessibility relation between them. Any modal sentence is thus equivalent (in a certain sense) to a non-modal sentence that explicitly discusses possible worlds and their inter-accessibility. Our translation of modal sentences to constraint sentences accords the accessibility relation a special status in the constraint logic.

The deductive system that reasons in the constraint logic is a hybrid system that combines ordinary deduction with special purpose methods for constraint processing. The construction of this deductive system is carried out within the the substitutional framework [Frisch, 1991] for the development of systems for hybrid reasoning. In this approach, a set of predicates are designated as constraint atoms. In the case of modal inference, the only predicate symbols in the designated set are K and E.

The input sentences to a hybrid reasoning system constructed within the substitutional framework are divided into two groups. One group of sentences, the S-sentences, contain no predicates other than those designated as constraint predicates. The S-sentences together form a theory called Σ. The other sentences, the A-sentences, are the constrained sentences. In the A-sentences, the predicates from the designated set occur only in the constraints. The only predicates that occur in the constraints are those from the designated set. In the modal case, the constraints on the A-sentences are conjunctions of K- and E-atoms.

The substitutional framework provides a method for transforming an ordinary deductive system into a hybrid deductive system in which the set of designated predicates are handled by a special-purpose reasoner. The overall deductive mechanism that reasons with the A-sentences invokes the special constraint processing mechanism to reason with the constraint predicates. All reasoning about the predicates from the designated set is done by the special purpose constraint processing mechanism.

The resolution and paramodulation rules of inference are used to illustrate the method by which a non-modal deductive system can be converted systematically to a constraint-deduction system that deals with constraints made up of K and E-atoms. So the final result will be closest to the resolution-based modal deduction systems of Ohlbach [1988] and of Auffray and Enjalbert [1989] and the development here of a resolution-based modal deductive method can be viewed as a reconstruction of these systems.

The framework is limited to logics whose possible world semantics restrict the accessibility relation to serial relations—some world is accessible from every world. The logics covered are those characterized by any combination of the reflexive, symmetric, transitive and Euclidian restrictions on the accessibility relation. Given the restriction to serial logics, this yields 9 distinct modal logics, including D, which is the minimal serial modal logic, T, for which the accessibility relation is required to be reflexive, $S4$, whose accessibility relation is required to be transitive as well as reflexive, and $S5$, which has an accessibility relation that is symmetric in addition to being reflexive and transitive.

In the course of this article, we illustrate the method being developed with a very simple version of the wise-man puzzle [Genesereth and Nilsson, 1987], which is commonly used as a test problem for formalisms involving knowledge and belief. The story is that there are some number of wise men who are told by the king that at least one of them has a white spot on his forehead. In reality, all of the wise men have white spots on their foreheads. Each wise man can see the others' foreheads, but not his own. In this version of the puzzle, there are two wise men named A and B. (1) A knows that if A does not have a white spot, B will know that A does not have a white spot. (2) A knows that B knows that either A or B has a white spot. B says that he does not know whether he has a white spot, and (3) A thereby knows that B does not know whether he has a white spot. The problem is to prove that (4) A knows that he has a white spot.

In a modal logic formulation of this puzzle, the modal operator $\Box$ is used to represent knowledge. It is necessary to introduce a distinct modal operator for each agent, which we call $\Box_A$ and $\Box_B$. The semantics must also be generalized to include an ac-

2 Background

The most general common instance, e, of a set, E, of atoms may be found with unification and characterizes the common instances of E: an atom is a common instance of E if and only if it is an instance of e.[1] Likewise, the most specific generalization (*msg*), g, of E characterizes the atoms that are more general than every atom in E: an atom is more general than every member of E if and only if it is more general than g. The ordinary generalization operation computes this most specific generalization (*msg*) of a set of atoms.

Algorithms for ordinary generalization have appeared elsewhere [Plotkin, 1970; Plotkin, 1971; Reynolds, 1970; Lassez *et al.*, 1988]. The following example illustrates the interesting aspects of ordinary generalization.

Example 1
Let $E = \{loves(clyde, mom(clyde)),$
$loves(jumbo, mom(jumbo))\}$.
Then $loves(x, mom(x))$ is an *msg* of E.

Notice that the common structure of the atoms in E is preserved in the *msg* of E. Notice also that the variable x repeats in the *msg* the same way that *clyde* and *jumbo* repeat in the first and second atoms, respectively, of E. The *msg* of a set of atoms whose sizes (number of variable and function symbol occurrences) sum to S can be computed in time $O(S)$.

Other researchers (for example, [Mitchell *et al.*, 1983]) have studied generalization with respect to *taxonomic background information*. Whereas generalizing "Clyde is gray" and "Jumbo is gray" yields the more general assertion, "Everything is gray," generalization with respect to the background information that Clyde and Jumbo are elephants yields the more specific result, "Elephants are gray." Sorted generalization combines ordinary generalization with the use of taxonomic background information. An important distinction between sorted generalization and such earlier work is that sorted generalization operates with respect to taxonomic theories of a very general form: first-order theories in which sorts act as monadic predicates. One result of such general taxonomic theories is that some sets of atoms have no most specific generalization. Rather, they may have a set of *minimal* generalizations, none more specific than another, and these minimal generalizations may differ significantly. Examples 3 and 4 in this section illustrate this aspect of sorted generalization. Most of the following material in this section repeats earlier work [Frisch and Page, 1990] that is needed as background for the remainder of this paper.

[1] Throughout the paper we use e and α, possibly subscripted, to denote atoms, which may be sorted or constrained atoms.

Sorted generalization operates on sorted atoms, which are like ordinary atoms except that they may contain *sorted variables*, variables that have *sort symbols* attached. (We also say that sorted terms are terms that may contain sorted variables.) The sorted variable x:ELEPHANT, for example, is restricted to range over the set of all elephants. Sorted generalization operates with respect to a taxonomic theory, or sort theory, that describes the sorts. A sort theory treats the sorts as monadic predicates. Several examples in the paper use the following sort theory:

$$\begin{aligned}\Sigma_1 = \{&\forall x\ \textsc{univ}(x),\ \textsc{gray}(mom(clyde)),\\ &\textsc{gray}(mom(jumbo)),\\ &\textsc{elephant}(clyde),\ \textsc{in-circus}(clyde),\\ &\textsc{elephant}(jumbo),\ \textsc{in-circus}(jumbo),\\ &\forall x\ \textsc{elephant}(x) \rightarrow \textsc{elephant}(mom(x)),\\ &\forall x\ \textsc{elephant}(x) \rightarrow \textsc{mammal}(x)\}.\end{aligned}$$

We are often interested in whether one sort is a subset of another sort according to a given sort theory Σ. τ' is a subsort of τ according to Σ if and only if $\Sigma \models \forall x(\tau'(x) \rightarrow \tau(x))$; we write $\tau' \preceq_\Sigma \tau$. If $\tau' \preceq_\Sigma \tau$ and $\tau \not\preceq_\Sigma \tau'$, then we write $\tau' \prec_\Sigma \tau$.

Where Σ is a sort theory, we say that one sorted atom is Σ-more general ($\geq_\Sigma$) than a second if and only if some substitution θ maps the first to the second, and θ respects the sorts of the variables. A substitution θ respects the sorts of variables relative to a sort theory Σ if and only if for every variable $x{:}\tau$, $\langle x{:}\tau\rangle\theta$ is a term t such that $\Sigma \models \overline{\forall}\tau(t)$.[2] If $e \geq_\Sigma e_1$ and $e \geq_\Sigma e_2$, we also say that e Σ-subsumes e_1 and e_2 and that e is a Σ-generalization of e_1 and e_2. If $e_1 \geq_\Sigma e_2$ and $e_2 \not\geq_\Sigma e_1$, we also write $e_1 >_\Sigma e_2$. The $\geq_\Sigma$ relation is particularly useful because under natural definitions for the universal and existential closures of sorted atoms [Frisch and Page, 1990], for any sorted atomic formulas, α_1 and α_2, and any sort theory, Σ: $\alpha_1 \geq_\Sigma \alpha_2$ if and only if $\{\overline{\forall}\alpha_1\} \cup \Sigma \models \overline{\forall}\alpha_2$ if and only if $\{\overline{\exists}\alpha_2\} \cup \Sigma \models \overline{\exists}\alpha_1$.

Example 2
$eats(x{:}\textsc{elephant}, nuts) \geq_{\Sigma_1} eats(clyde, nuts)$.

To see this notice that for $\theta_1 = \{clyde/x{:}\textsc{elephant}\}$,

$$eats(x{:}\textsc{elephant}, nuts)\theta = eats(clyde, nuts).$$

Notice further that Σ_1 entails that clyde is an elephant. On the other hand,

$$eats(x{:}\textsc{elephant}, nuts) \not\geq_{\Sigma_1} eats(fido, nuts).$$

To see this, notice that any substitution mapping $eats(x{:}\textsc{elephant}, nuts)$ to $eats(fido, nuts)$ must map

[2] For any expression ψ, $\overline{\forall}\psi$ denotes the universal closure of ψ, the result of universally quantifying all free variables of ψ. Similarly, $\overline{\exists}\psi$ denotes the existential closure of ψ.

x:ELEPHANT to *fido*. But Σ_1 does not entail that fido is an elephant.

Example 3
Let $E = \{eats(clyde,nuts), eats(jumbo,nuts)\}$.
E has two maximally-specific sorted generalizations with respect to Σ_1:

$eats(x$:ELEPHANT$, nuts)$, and
$eats(y$:IN-CIRCUS$, nuts)$.

E has a most specific unsorted generalization:

$eats(x, nuts)$.

We would like to use a Σ-generalization to characterize the sorted atoms that are Σ-more general than any given set of sorted atoms. But Example 3 shows that there may be no most specific Σ-generalization: neither $eats(x$:ELEPHANT$,nuts)$ nor $eats(y$:IN-CIRCUS$,nuts)$ is more specific, with respect to Σ, than the other. An analogous characterization therefore requires a set of maximally specific Σ-generalizations. Specifically, the goal of sorted generalization is to find a *complete set of incomparable Σ-generalizations* (or CIG_Σ) for a given set of sorted atoms. Σ-generalizations are incomparable if none is Σ-more general than another. A set G of maximally specific, incomparable Σ-generalizations of a set E of sorted atoms is a CIG_Σ of E if and only if any other Σ-generalization of E is Σ-more general than some sorted atom in G. Every set (even infinite) of sorted atoms has a finite CIG_Σ. For some sets the CIG_Σ is empty.

The following example illustrates an additional distinction between sorted and ordinary generalization, which a sorted generalization algorithm must accommodate. In Example 4 "*mom*" is a function symbol, so E is a set of terms.

Example 4
Let $E = \{mom(clyde), mom(jumbo)\}$.
Then a CIG_{Σ_1} of E is

$\{y$:GRAY$, mom($x:ELEPHANT$), mom(z$:IN-CIRCUS$)\}$,

and $mom(x)$ is a most specific unsorted generalization.

In Example 4, y:GRAY Σ_1-subsumes neither $mom(x$:ELEPHANT$)$ nor $mom(z$:IN-CIRCUS$)$. This contrasts with the unsorted case, where a variable always subsumes any other term. We refer to a variable that Σ-subsumes all members of a set of (possibly sorted) terms as a *variable generalization* of that set. We refer to a non-variable sorted atom that Σ-subsumes all members of a set as a *structured generalization*. A sorted generalization algorithm cannot assume that a variable generalization of a set of (sorted) terms Σ-subsumes every structured generalization of that set. It must therefore compare the variable generalizations with the structured generalizations.

Because variable generalizations need not Σ-subsume structured generalizations, members of a CIG_Σ may have different structures. In the CIG_{Σ_1} of Example 4, one generalization is a variable while the other two are built from the function symbol "*mom*." Such structural differences can become more remarkable as the expressions being generalized grow more complex. That variable generalizations need not Σ-subsume structured generalizations is a result of *function polymorphism*. A function is polymorphic if terms built from that function's symbol may denote members of different sorts based on the arguments in the terms. *mom* is a polymorphic function, because whether a term built from "*mom*" denotes a member of the sort GRAY depends on the argument to *mom*. If the argument is *clyde* or *jumbo*, the term denotes something gray; otherwise it does not. If *mom* were instead a monomorphic function, that is, if all moms were known to be gray, the variable y:GRAY would Σ_1-subsume any structured generalization.

Elsewhere [Frisch and Page, 1990] we present an algorithm that computes sorted generalization given oracles for two kinds of *taxonomic queries*. These two kinds of queries are actually equivalent to a single kind of taxonomic query that, given a sort theory, Σ, and a set of sorted terms, $\{t_1, ..., t_n\}$, asks for the minimal (under the $\preceq_\Sigma$ ordering) sorts in the set $\{\tau \mid \Sigma \models \forall \tau(t_i)$ for all $1 \leq i \leq n\}$. Therefore an oracle, or decision procedure, for these taxonomic queries is sufficient for computation of sorted generalization. An oracle for these queries is in fact necessary as well. In general such an oracle cannot be computed, but restrictions can be placed on the sort theory so that it can be. Unfortunately, the size of a CIG_Σ can be exponential in the sizes of the sorted atoms generalized, so generalization can require exponential time and space. Though we can represent the CIG_Σ compactly, in polynomial size, computing even a compactly represented CIG_Σ is NP-hard.

We close this section with a new result. Probably the most obvious application of generalization is inductive learning. Two widely-appreciated models of successful inductive learning are *polynomial pac-learning* [Valiant, 1984; Blumer *et al.*, 1989] and *polynomial learning by equivalence queries* [Angluin, 1988a; Angluin, 1988b]. Learning in the second model implies learning in the first [Angluin, 1988b]. Can sorted atoms be learned in these models given efficient response to taxonomic queries? Before answering this question, we first define these inductive learning models as they apply to sorted atoms.[3]

For any two sorted atoms, α_1 and α_2, and any sort theory, Σ, α_1 is said to *cover* α_2 with respect to Σ if and only if $\alpha_1 \geq_\Sigma \alpha_2$. Two sorted atoms are equivalent

[3]The definitions are based on definitions given in [Angluin, 1988a; Angluin, 1988b; Blumer *et al.*, 1989].

with respect to Σ if and only if they cover the same sorted atoms with respect to Σ. A sorted atom α_2 is labeled as a *positive example* by another sorted atom, α_1, and a sort theory, Σ, if α_1 covers α_2 with respect to Σ. Otherwise, α_2 is labeled as a *negative example* by α_1 and Σ. Each sorted atom has an associated size, which is the number of variable and function symbol occurrences in the atom.

Definition 5 *An* equivalence query *asks whether a particular sorted atom (the* predicted *atom) is equivalent to an unknown target sorted atom with respect to the sort theory Σ. An oracle for equivalence queries answers* yes *to any given query if the predicted atom is equivalent to the target with respect to Σ; it answers* no *otherwise. The oracle gives a* counterexample *if the predicted atom is not equivalent to the target. A* counterexample *to an equivalence query is any sorted atom labeled differently by the predicted and target concepts with respect to Σ.*

Definition 6 *An algorithm A is a* polynomial-time learning algorithm for sorted atoms with fixed Σ *if and only if for any fixed sort theory Σ, for which any taxonomic query is answered in time polynomial in the size of the query, there exists a polynomial $p(n, m)$ such that: for any target sorted atomic formula α, when A is run with an oracle to answer equivalence queries for α with respect to Σ, A halts and outputs a sorted atom that is equivalent to α. Moreover, at any point during the run, the time used by A to that point is bounded by $p(n, m)$, where n is the size of α, and m is the length of the longest counterexample seen to that point. Sorted atoms, with fixed Σ, are* polynomially learnable by equivalence queries *if and only if there exists such an algorithm A.*

Definition 7 *An algorithm A is a* polynomial-time pac-learning algorithm for sorted atoms with fixed Σ *if and only if for any fixed sort theory, Σ, for which the response to any taxonomic query is returned in time polynomial in the size of the query, A has the following property. For all $0 < \epsilon, \delta < 1$, and $n \geq 1$ there exists a sample size $m(\epsilon, \delta, n)$, polynomial in $1/\epsilon$, $1/\delta$, and n, such that for all sorted atoms, α, of size n or less, and all probability distributions, P, on the sorted atoms, given a random sample of $m(\epsilon, \delta, n)$ sorted atoms drawn independently according to P and labeled according to α and Σ, A produces, with probability at least $1 - \delta$, a sorted atom, α', such that the probability that a given sorted atom drawn according to P is labeled differently, with Σ, by α and α' is at most ϵ. Sorted atoms are* polynomially pac-learnable for fixed Σ *if and only if such an algorithm A exists.*

Applying one result of Computational Learning Theory [Pitt and Valiant, 1988], we find that learning sorted atoms with respect to a given sort theory, Σ, in these models is possible only if a (possibly randomized, for pac-learning) polynomial-time algorithm for the following problem exists. Given a set of sorted atoms, each labeled as either a positive or negative example, determine whether there exists a sorted atom, α, that covers, with Σ, all of the positive examples but none of the negative examples. Such a sorted atom is said to be *consistent with the examples*, and we call the problem of determining whether such a sorted atom exists the *consistency problem* for sorted atoms. We have noted that sorted generalization requires exponential time, and that even with a compact representation it requires exponential time assuming $P \neq NP$. The following result tells us that this complexity carries over to the consistency problem, and so to both pac-learning and learning by equivalence queries if we assume $R \neq NP$.[4]

Theorem 8 *Sort theories Σ exist for which the consistency problem is NP-hard, even though any taxonomic query about Σ is answered in time polynomial in the size of the query.*

Therefore, assuming $R \neq NP$, no algorithms exist that polynomially pac-learn sorted atoms or polynomially learn sorted atoms by equivalence queries.

3 A More General Language

In some cases generalization and inductive learning become easier if we broaden the language of acceptable generalizations. In this section we extend the language of both observations and acceptable generalizations. This extension yields efficient generalization and even positive results (provided in Section 4) for polynomial pac-learning and polynomial learning by equivalence queries. We extend the language in stages, eventually replacing sorts, which may be viewed as monadic constraints on variables, with arbitrary constraints among terms. Generalization with constraint information may be viewed as a dual[5] of constraint unification, a form of unification that is useful in constraint deduction and constraint logic programming.

3.1 Extended Sorted Generalization

We begin our extension of the language by changing the syntax of sorted atoms. Following work in constraint logic, we use separate specifications for the structure of a sorted atom and the sorts, or constraints, on its variables. In the new syntax, for example, the sorted atom

$$loves(x{:}\text{ELEPHANT}, mom(z{:}\text{IN-CIRCUS}))$$

is written as

[4] That is, if we assume the NP-complete problems cannot be solved in random polynomial time.

[5] We use "dual" in a non-technical sense.

$$loves(x, z)/\text{ELEPHANT}(x) \wedge \text{IN-CIRCUS}(z).$$

The structure of the sorted atom appears first, followed by a slash and then the specifications of the sorts. The expression that represents the structure of the sorted atom is called the *head* of the sorted atom. The conjunction of sort specifications is called the *constraint* of the sorted atom. Sorted terms are like sorted atoms except the head is a term.

Having changed the syntax in this way, we may constrain any terms in a sorted atom, rather than the variables only, to belong to a given sort. Thus sorted atoms have not only sorted variables, but sorted terms in general. For example, we may now constrain $mom(z)$ to be a lawyer. Thus

$$loves(x, mom(z))/\text{ELEPHANT}(x) \wedge \text{IN-CIRCUS}(z) \wedge \text{LAWYER}(mom(z))$$

is a legal sorted atom. It says "x, which is an elephant, loves the mom of z, where z is in a circus and the mom of z is a lawyer." Note that we require all terms that appear in the constraint of a sorted atom to appear in the head as well. Thus, for example,

$$loves(x, z)/\text{LAWYER}(father(z))$$

is not a legal sorted atom. Absence of this requirement leads to greatly increased computational costs in generalization.

We further expand the definition of sorted atoms to allow terms to be constrained with multiple sorts. For example,

$$loves(x, z)/\text{ELEPHANT}(x) \wedge \text{ELEPHANT}(z) \wedge \text{IN-CIRCUS}(z)$$

becomes a sorted atom. It says "x, which is an elephant, loves z, which is an elephant and a circus animal."

Section 2 takes the size of a sorted atom to be the number of variable and function symbol occurrences it contains. Now that we have more general constraints on terms, it is useful to distinguish between the *head size* of a sorted atom, which is the number of variable and function symbol occurrences in the head, and the *total size* of a sorted atom, which is the head size plus the number of variable, function symbol, and constraint predicate occurrences in the constraint of the sorted atom.

Having expanded the definition of sorted atoms, we must expand our semantic rules for quantification. Let $\alpha = A/a_1 \wedge ... \wedge a_n$ be a sorted atom. Because we are interested in only the universal and existential closures of sorted atoms, it is sufficient to specify that $\overline{\forall}\alpha$ is equivalent to $\overline{\forall}(a_1 \wedge ... \wedge a_n \rightarrow A)$ and $\exists\alpha$ is equivalent to $\exists(a_1 \wedge ... \wedge a_n \wedge A)$. Note that a universally-closed sorted atom is equivalent to a definite clause whose body (the constraint) is built of sort symbols and terms that appear in its head.

We would like to again base our generalization algorithm on a Σ-more general ($\geq_\Sigma$) relation. Recall that under the original definition of sorted atoms, one sorted atom is Σ-more general than a second if some substitution maps the first to the second and respects the sorts of all variables with respect to Σ. Recall also that a substitution, θ, respects the sorts of all variables if for every variable $x{:}\tau$, θ maps $x{:}\tau$ to a term t for which $\Sigma \models \overline{\forall}\tau(t)$. But under the new definition of sorted atoms, arbitrary terms, rather than variables only, may be assigned sorts. Therefore, insisting only that a substitution must respect the sorts of the variables is too weak. We instead want to require substitutions that respect the sorts of all terms.

Our new syntax for sorted atoms lets us phrase the definition of $\geq_\Sigma$ another way. Let $\alpha = A/a_1 \wedge ... \wedge a_n$ and $\alpha' = A'/a_1' \wedge ... \wedge a_m'$ be sorted atoms. We say that $\alpha \geq_\Sigma \alpha'$ if a substitution θ exists for which $A\theta = A'$ and $\Sigma \models \overline{\forall}(a_1' \wedge ... \wedge a_m' \rightarrow (a_1 \wedge ... \wedge a_n)\theta)$. In the cases where α and α' are sorted atoms under the original definition of sorted atoms—that is, where sorts are specified on only variables in the atoms, and at most one sort is specified on each variable—this definition of $\geq_\Sigma$ is equivalent to the original one. Note that this relation may be applied to sorted terms as well.

Example 9

$loves(x,y)/\text{MAMMAL}(x) \wedge \text{MAMMAL}(y)$
$\geq_{\Sigma_1} loves(z,mom(z))/\text{ELEPHANT}(z).$

To see this, notice first that for any substitution θ that maps x to z and y to $mom(z)$:

$$loves(x, y)\theta = loves(z, mom(z)).$$

Second, note that

$$(\text{MAMMAL}(x) \wedge \text{MAMMAL}(y))\theta = \text{MAMMAL}(z) \wedge \text{MAMMAL}(mom(z)).$$

Because Σ_1 entails that the mom of an elephant is an elephant and that all elephants are mammals,

$$\Sigma_1 \models \overline{\forall}(\text{ELEPHANT}(z) \rightarrow (\text{MAMMAL}(z) \wedge \text{MAMMAL}(mom(z)))).$$

The following is an example involving an expression that is a sorted atom under the new definition but not under the old one.

Example 10

$loves(x, mom(x))/\text{ELEPHANT}(x) \wedge \text{IN-CIRCUS}(x) \wedge \text{GRAY}(mom(x))$
$\geq_{\Sigma_1} loves(clyde, mom(clyde)).$

For any substitution θ that maps x to clyde,

$$loves(x, mom(x))\theta = loves(clyde, mom(clyde)).$$

Note further that

$$\begin{aligned}&(\text{ELEPHANT}(x) \wedge \text{IN-CIRCUS}(x) \\ &\qquad \wedge \text{GRAY}(mom(x)))\theta \\ &= \text{ELEPHANT}(clyde) \wedge \text{IN-CIRCUS}(clyde) \\ &\qquad \wedge \text{GRAY}(mom(clyde)).\end{aligned}$$

We can easily verify that

$$\begin{aligned}\Sigma_1 \models\ &\text{ELEPHANT}(clyde) \wedge \text{IN-CIRCUS}(clyde) \\ &\wedge \text{GRAY}(mom(clyde)).\end{aligned}$$

Finally, as a negative example of the $\geq_\Sigma$ relation,

Example 11
$loves(x, mom(x))/\text{ELEPHANT}(x) \wedge \text{IN-CIRCUS}(x)$
$\not\geq_{\Sigma_1} loves(x, mom(x))/\text{MAMMAL}(x) \wedge \text{GRAY}(mom(x))$.

The relation fails to hold because

$$\begin{aligned}\Sigma_1 \not\models \forall((&\text{MAMMAL}(x) \wedge \text{GRAY}(mom(x))) \\ &\rightarrow (\text{ELEPHANT}(x) \wedge \text{IN-CIRCUS}(x))).\end{aligned}$$

Having expanded the definitions of sorted atoms and the $\geq_\Sigma$ relation, any set of sorted atoms has a singleton CIG_Σ. That is, any set of sorted atoms has a most specific Σ-generalization (msg_Σ). To see why, let's rework Example 4 using the extended definitions of sorted atoms and sorted terms.

Example 12
Let E be $\{mom(clyde), mom(jumbo)\}$.
Then an msg_{Σ_1} of E is

$$\begin{aligned}&mom(x)/\text{ELEPHANT}(x) \wedge \text{IN-CIRCUS}(x) \wedge \\ &\qquad \text{GRAY}(mom(x)).\end{aligned}$$

The CIG_{Σ_1} in Example 4 has multiple generalizations for two reasons. The first reason is that two minimal sorts, rather than one, contain both *clyde* and *jumbo*. Therefore, under the original definition of sorted atoms, we obtain two minimal generalizations of *clyde* and *jumbo*—x:ELEPHANT and x:IN-CIRCUS—which lead to two minimal generalizations of $mom(clyde)$ and $mom(jumbo)$. Under the new definition of sorted atoms, we can attach multiple sorts to a variable, thereby obtaining one minimal generalization of *clyde* and *jumbo*—a variable, x, belonging to both ELEPHANT and IN-CIRCUS sorts. Notice that both sorts are attached to x in Example 12. The second reason for multiple minimal generalizations in Example 4 is that both $mom(clyde)$ and $mom(jumbo)$ belong to the sort GRAY, while neither $mom(x$:ELEPHANT$)$ nor $mom(x$:IN-CIRCUS$)$ belong to that sort. Therefore, a variable of the sort GRAY is a third minimal generalization. Under the new definition of sorted atoms, we can always add sorts on an entire non-variable term to make it more specific, with respect to the sort theory, than any given variable. We do so in Example 12 by constraining $mom(x)$ to belong to the sort GRAY, thus making it more specific than a variable of sort GRAY.

The Extended Sorted Generalization (Extended SG) Algorithm (Figure 1) computes the msg_Σ of any finite set of extended sorted atoms. The presentation of the algorithm requires one additional definition. Given an expression e_1 and a particular occurrence γ of a term in e_1, the Extended SG Algorithm needs to be able to select from another expression e_2 the term t that occurs at the same position in e_2 as does γ in e_1. We say that t is *the term in e_2 corresponding to γ in e_1*. For example, if e_1 is $P(f(a), a)$ and e_2 is $P(f(x), f(x))$, then x is the term in e_2 corresponding to the first occurrence of a in e_1, and $f(x)$ is the term in e_2 corresponding to the second occurrence of a in e_1. More precisely we define the term in e_2 corresponding to γ in e_1 to be t if, and only if:

- γ is e_1 and t is e_2, or
- e_1 is of the form $f(s_1, \ldots, s_n)$ and e_2 is of the form $f(u_1, \ldots, u_n)$, where f is a function symbol or predicate, and γ is in s_i and t is the term in u_i corresponding to γ in s_i.

If one occurrence of a term t_1 in an expression e corresponds to a term t_2 in an expression e', then so does every other occurrence of t_1 in e. In this case we sometimes speak of the term in e' that corresponds to a given term, rather then term occurrence, in e.

The Extended SG Algorithm interacts with a sort theory Σ only through the following oracle. For all $1 \leq i \leq n$, let $a_{i,1} \wedge \ldots \wedge a_{i,m_i}$ be any constraint and let t_i be any term. The oracle returns the members of the set $\{\tau \mid \Sigma \models \forall(a_{i,1} \wedge \ldots \wedge a_{i,m_i} \rightarrow \tau(t_i))$ for all $1 \leq i \leq n\}$ that are minimal under the $\preceq_\Sigma$ ordering. This oracle is stronger than the analogous one used in sorted generalization as described in Section 2. Another way of specifying the oracle of Section 2 is that it is a restriction of the oracle used by the Extended SG Algorithm to cases where each constraint specifies sorts for only variables and specifies at most one sort for each variable. The oracle used by the Extended SG Algorithm is in fact necessary for sorted generalization in the extended language if we wish to avoid constraints with redundancy:[6] given any algorithm for extended sorted generalization that avoids redundancy in the constraints, it can be used to compute the oracle. An obvious, weaker oracle is necessary and sufficient if we allow redundancy in constraints.

Theorem 13 *Let Σ be a sort theory such that any taxonomic query of size s (s variable, function symbol, and constraint predicate occurrences) about Σ is made*

[6] A constraint contains redundancy if it contains sort specifications $\tau_1(t)$ and $\tau_2(t)$, where $\tau_1 \preceq_\Sigma \tau_2$.

Extended Sorted Generalization (Extended SG) Algorithm
Input: n sorted atoms, $\alpha_i = A_i/a_{i,1} \wedge ... \wedge a_{i,m_i}$ for $1 \leq i \leq n$, and a sort theory Σ.
Output: An msg_Σ of α_1, ..., α_n.

1. Let Head = Unsorted-Generalization$(A_1, ..., A_n)$.
2. If Head = *nil*, then return *nil*.
3. Initialize Constraint-Set to the empty set.
4. For each term u in Head:
   a. Let t_i be the corresponding term in A_i, for all $1 \leq i \leq n$.
   b. Let Min-Sorts be the minimal (under the $\preceq_\Sigma$ ordering) members of $\{\tau \mid \Sigma \models \forall(a_{i,1} \wedge ... \wedge a_{i,m_i} \rightarrow \tau(t_i))$ for all $1 \leq i \leq n\}$.
   c. For each sort τ in Min-Sorts, add the constraint formula $\tau(u)$ to Constraint-Set.
5. Return the sorted atom whose head is Head and whose constraint is the conjunction of the constraint formulas in Constraint-Set.

Figure 1: The Extended Sorted Generalization (SG) Algorithm

and answered in time at most $Q(s)$.[7] *The Extended Sorted Generalization Algorithm computes an msg_Σ of a finite set of extended sorted atoms whose total sizes sum to S and which contains at least one atom of head size no greater than l in time $O(lQ(S))$.*

Proof:[Time Complexity] The time for the ordinary generalization step is $O(H)$ where H is the sum of the head sizes of the sorted atoms being generalized. The time for computing term correspondences (for example, establishing pointers from each term in the ordinary generalization, the *msg* of the heads, to the corresponding terms) is also $O(H)$. The size of the ordinary generalization can be no greater than the least head size among the sorted atoms being generalized. Therefore, there are at most l terms in the ordinary generalization. For each of these l terms, the algorithm uses the corresponding terms to build a taxonomic query. Because term correspondences have already been computed, any taxonomic query can be built in time linear in the size of the query, which can be at most S. The time for a query to be made and answered must be at least linear in the size of the query and the number of sorts in Σ (the size of the largest answer), so the term $Q(S)$ dominates the time spent in building the query. Because $S \geq H$, this term also dominates the time spent in ordinary generalization and in computing corresponding terms. □

[7] For some sort theories, it may be desirable to measure the time complexities of queries in a more detailed way, such as by dividing the size of a query into the sizes of the terms and the sizes of the constraints used in the query.

3.2 Constraint Generalization

Many interesting concepts require constraints *among* terms. Our present language can only represent monadic constraints, or sorts, *on* single terms. Thus sorted atoms as presently defined cannot represent, for example, the concept that electric current flows from any one object x to any other object y given the constraints that x is metallically connected to y and that x has a higher electrical potential than does y.

We overcome this shortcoming as follows. In place of a set of distinguished monadic predicates, which we have been calling sorts, we use a set of distinguished predicates of arbitrary arity. We call these predicates *constraint predicates*. Analogous to sort theories, a constraint theory is any theory whose only predicates are constraint predicates. Rather than sorted atoms, we now have constrained atoms, or atoms with constraints *among* terms in addition to monadic constraints, or sorts, *on* terms. The definition of $\geq_\Sigma$ remains unchanged.

Following are several examples of the $\geq_\Sigma$ relation for constrained atoms. Some of these examples use the following constraint theory, Σ_2.

$\Sigma_2 = \{$BIGGER(*son*(*jumbo*),*son*(*clyde*)),
BIGGER(*son*(*fred*),*son*(*joe*)),
BIGGER(*jumbo*,*clyde*),
BIGGER(*fred*,*joe*)},
ELEPHANT(*clyde*), ELEPHANT(*jumbo*),
HUMAN(*fred*), HUMAN(*joe*),
$\forall x \forall y$ ELEPHANT$(x) \wedge$ HUMAN$(y) \rightarrow$
BIGGER$(x, y)\}$.

Example 14
intimidates(*son*(x_1), *son*(y_1))/

Constraint Generalization Algorithm
Input: n constrained atoms, $\alpha_i = A_i/a_{i,1} \wedge ... \wedge a_{i,m_i}$ for $1 \leq i \leq n$, and a constraint theory Σ.
Output: An msg_Σ of α_1, ..., α_n.

1. Let Head = Ordinary-Generalization(A_1, ..., A_n).

2. If Head = *nil*, then return *nil*.

3. Initialize Constraint-Set to the empty set.

4. Let r be the maximum of the arities of all constraint predicate in Σ.
   For all $1 \leq l \leq r$ and each tuple (each permutation of each set) $\langle u_1, ..., u_l \rangle$ of l terms in Head:
   a. Let $\langle t_{i,1}, ..., t_{i,l} \rangle$ be the tuple of corresponding terms in A_i, for all $1 \leq i \leq n$.
   b. Let Min-Constraint-Predicates be the minimal (under the $\preceq_\Sigma$ ordering) members of $\{\tau \mid \tau$ has arity l, and $\Sigma \models \overline{\forall}(a_{i,1} \wedge ... \wedge a_{i,m_i} \rightarrow \tau(t_{i,1}, ..., t_{i,l}))$ for all $1 \leq i \leq n\}$.
   c. For each constraint predicate τ in Min-Constraint-Predicates, add the constraint formula $\tau(u_1, ..., u_l)$ to Constraint-Set.

5. Return the constrained atom whose head is Head and whose constraint is the conjunction of the constraint formulas in Constraint-Set.

Figure 2: The Constraint Generalization Algorithm

$$\begin{aligned}
&\textsc{bigger}(son(x_1), son(y_1)) \\
&\geq_{\Sigma_2} \\
&intimidates(son(x_2), son(y_2))/ \\
&\quad \textsc{bigger}(x_2, y_2) \wedge \textsc{bigger}(son(x_2), son(y_2)) \\
&\geq_{\Sigma_2} \\
&intimidates(son(jumbo), son(clyde)).
\end{aligned}$$

To verify that the first relation holds, first note that any substitution θ mapping x_1 to x_2 and y_1 to y_2 also maps $intimidates(son(x_1),son(y_1))$ to $intimidates(son(x_2),son(y_2))$. We then need to see if Σ_2 entails

$$\begin{aligned}
&\overline{\forall}(\textsc{bigger}(x_2, y_2) \wedge \textsc{bigger}(son(x_2), son(y_2)) \\
&\quad \rightarrow (\textsc{bigger}(son(x_1), son(y_1)))\theta).
\end{aligned}$$

It does, because

$$\begin{aligned}
&\textsc{bigger}(son(x_1), son(y_1))\theta \\
&= (\textsc{bigger}(son(x_2), son(y_2))),
\end{aligned}$$

and Σ_2 entails

$$\begin{aligned}
&\overline{\forall}(\textsc{bigger}(x_2, y_2) \wedge \textsc{bigger}(son(x_2), son(y_2)) \\
&\quad \rightarrow \textsc{bigger}(son(x_2), son(y_2))).
\end{aligned}$$

To see that the second relation holds, consider a substitution θ that maps x_2 to jumbo and y_2 to clyde. First, notice that

$$\begin{aligned}
&intimidates(son(x_2),son(y_2))\ \theta = \\
&intimidates(son(jumbo),son(clyde)).
\end{aligned}$$

Second, note that

$$\begin{aligned}
&(\textsc{bigger}(x_2, y_2) \wedge \\
&\quad \textsc{bigger}(son(x_2), son(y_2)))\theta \\
&= \textsc{bigger}(jumbo, clyde) \wedge \\
&\quad \textsc{bigger}(son(jumbo), son(clyde)),
\end{aligned}$$

and that Σ_2 entails

$$\begin{aligned}
&\textsc{bigger}(jumbo,clyde) \wedge \\
&\textsc{bigger}(son(jumbo),son(clyde)).
\end{aligned}$$

The only change required to the Extended SG Algorithm to deal with the more general language is in step 4, that of computing the constraints. Rather than compute the sorts to place on each term in the head of the generalization, the algorithm must instead determine which constraints to place on each tuple of terms in the head. The algorithm that results from this modification is the Constraint Generalization Algorithm (Figure 2). Note that the algorithm need not consider tuples of more than r terms, where r is the maximum of the arities of all constraint predicates in Σ.

For the presentation of the algorithm, it is useful to extend the $\preceq_\Sigma$ relation defined on sorts to apply to arbitrary constraint predicates of the same arity. For n-ary constraint predicates τ_1 and τ_2, we say that $\tau_1 \preceq_\Sigma \tau_2$ if and only if $\Sigma \models \overline{\forall}(\tau_1(x_1, ..., x_n) \rightarrow \tau_2(x_1, ..., x_n))$.

The Constraint Generalization Algorithm uses constraint queries, rather than taxonomic queries, to interact with the constraint theory. The algorithm needs to know the maximum of the arities of all constraint predicates in Σ. It also needs the following oracle, which is stronger than that used by the Extended SG

Algorithm. For all $1 \leq i \leq n$, let $a_{i,1} \wedge ... \wedge a_{i,m_i}$ be a constraint and let $\langle t_{i,1}, ..., t_{i,j} \rangle$ be a tuple of j terms, where j is less than the maximum of the arities of all constraint predicates in Σ. The oracle returns the members of the set $\{\tau \mid \tau$ has arity l, and $\Sigma \models \overline{\forall}(a_{i,1} \wedge ... \wedge a_{i,m_i} \rightarrow \tau(t_{i,1}, ..., t_{i,l}))$ for all $1 \leq i \leq n\}$ that are minimal under the $\preceq_\Sigma$ ordering. This oracle is in fact necessary for computation of constraint generalization by any algorithm if we want to exclude redundancy *among constraint predicates of the same arity.* Again, if we do not wish to exclude such redundancy, an obvious, weaker oracle is necessary and sufficient. Because the algorithm does not compare constraint predicates of different arity, the constraint on the constrained atom it returns may contain some redundancy, but the constrained atom is still an msg_Σ. The following examples shows msg_Σs the algorithm computes.

Example 15
Let E be
$\{intimidates(son(jumbo), son(clyde)),$
$intimidates(son(fred), son(joe))\}$.
An msg_{Σ_2} of E is:
$intimidates(son(x_2), son(y_2))/$
$\text{BIGGER}(x_2, y_2) \wedge$
$\text{BIGGER}(son(x_2), son(y_2))$.

Example 16
Let E be
$\{intimidates(clyde, y_1)/\text{HUMAN}(y_1),$
$intimidates(x_1, fred)\}/\text{ELEPHANT}(x_1)\}$.
An msg_{Σ_2} of E is:
$intimidates(x_2, y_2)/\text{BIGGER}(x_2, y_2) \wedge$
$\text{ELEPHANT}(x_2) \wedge \text{HUMAN}(y_2)$.

Theorem 17 *Let Σ be a constraint theory such that the maximum of the arities of all constraint predicates in Σ is r, and any constraint query about Σ having size s is made and answered in time $Q(s)$. The Constraint Generalization Algorithm computes an msg_Σ of n constrained atoms whose total sizes sum to S and which contains at least one atom with head size no greater than l in time $O(rl^r Q(rS))$.*

Proof:[Time Complexity] The analysis is the same as that for Theorem 13 except for two points. First, the size of a constraint query can be almost as large as rS, rather than S, because the query uses tuples of up to r terms, some of which may appear as subterms of others.[8] Second, rather than querying the oracle for each term in the ordinary generalization, the algorithm must query the oracle for each tuple of 1 to r terms, the number of which is bounded by rl^r. □

[8]Suppose, for example, that r is 2, and the constrained atoms being generalised have empty constraints (so $H = S$). Suppose also that the heads of the atoms each have one argument (of size S/n), each of which consists of a function applied to an argument of size $(S/n) - 1$. Then there is a query of size $2S - n$. So if three constrained atoms are being generalised, the query has size $rS - 3$, where $r = 2$.

4 Applications

Applications of sorted generalization to abduction, induction, analogical reasoning, and knowledge base vivification are based on semantic properties of the $\geq_\Sigma$ relation (Section 2 and [Frisch and Page, 1990]). These same properties hold for the extended $\geq_\Sigma$ relation for extended sorted generalization and constraint generalization.

Theorem 18 *Let Σ be a constraint theory and let α_1 and α_2 be constrained atoms. $\alpha_1 \geq_\Sigma \alpha_2$ if and only if $\{\overline{\forall}\alpha_1\} \cup \Sigma \models \overline{\forall}\alpha_2$.*

Theorem 19 *Let Σ be a constraint theory and let α_1 and α_2 be constrained atoms. $\alpha_1 \geq_\Sigma \alpha_2$ if and only if $\{\overline{\exists}\alpha_2\} \cup \Sigma \models \overline{\exists}\alpha_1$.*

Theorem 18 justifies the use of constraint generalization for abduction and induction. Theorem 19 makes constraint generalization applicable to knowledge base vivification. Constraint generalization is applicable to analogical reasoning, based on both theorems. Rather than repeat arguments made elsewhere [Frisch and Page, 1990], we further investigate the application to induction. More specifically, we ask whether the Constraint Generalization Algorithm can be used for polynomial learning by equivalence queries and polynomial pac-learning. The definitions of these learning models for constrained atoms are the same as for sorted atoms (Section 2), except Σ may be any constraint theory, rather than sort theory, and queries are constraint queries, rather than taxonomic queries.

If we add to the set of constrained atoms a weakest element that (with any constraint theory Σ) covers no constrained atoms, then the following algorithm polynomially learns constrained atoms by equivalence queries with respect to any fixed constraint theory Σ. The algorithm first predicts the weakest element. If it is correct, the algorithm is finished. If it is incorrect, the only counterexamples are constrained atoms labeled as positive examples. One positive example is received in response to the prediction. From this point on, the algorithm predicts the msg_Σ of the positive examples received thus far. Because the msg_Σ is the weakest constrained atom Σ-more general than the positive examples seen thus far, it is always weaker than or equivalent to the target constrained atom. Therefore the algorithm receives only positive examples as counterexamples.

Because the Constraint Generalization Algorithm runs in time polynomial in the sizes of the constrained atoms being generalized, each prediction can be computed in time polynomial in the sizes of the counterex-

amples seen thus far. The following theorem (the proof of which is in the appendix) shows that for fixed Σ the algorithm makes at most a number of mistakes polynomial in the size of the first counterexample the algorithm receives. After this many mistakes, the algorithm predicts correctly. This result completes the argument that the algorithm polynomially learns constrained atoms by equivalence queries for any fixed Σ, since the total time for the algorithm to reach the correct prediction is the product of the number of mistakes and the maximum time to compute a prediction.

Theorem 20 *Let C be the number of constraint predicates in Σ, and let r be the maximum of the arities of all constraint predicates in Σ. The maximum chain $\alpha_1 \geq_\Sigma \ldots \geq_\Sigma \alpha_m$ of constrained atoms of head size at most n has length (m) at most $Cn^r + n + 1$.*

Because each prediction of the learning algorithm is strictly Σ-more general than the previous prediction, and the maximum length of such a chain of predictions is Cn^r+n+1, the algorithm makes at most Cn^r+n+1 prediction mistakes (counting the original prediction of the element that covers no constrained atoms). C and r are constant for a fixed constraint theory, so the number of prediction mistakes is polynomial in n.

As Section 2 notes, polynomial learning by equivalence queries implies polynomial pac-learning. More specifically, any algorithm that computes a constrained atom consistent with a set of constrained atoms, where each atom is labeled as a positive or negative example, in time polynomial in the sizes of the examples can be used to pac-learn constrained atoms. The Constraint Generalization Algorithm can be used for such computations with respect to any fixed constraint theory, Σ, for which we have polynomial-time response to constraint queries. To do so, first use the Constraint Generalization Algorithm to compute the msg_Σ of the positive examples. If any constrained atom is consistent with the set of examples, the msg_Σ of the positive examples is consistent. Then check whether the msg_Σ covers any negative examples. This check can be done in polynomial time using the same constraint queries used by the Constraint Generalization Algorithm. If the msg_Σ covers no negative examples, then it is consistent with the set of examples. Otherwise, no constrained atom is consistent with the set of examples; that is, the target concept is not a constrained atom.

The mistake bound for polynomial learning by equivalence queries gives us a sufficient number of examples to use for a given desired accuracy and confidence in pac-learning [Angluin, 1988b]. The next result, which is also achieved using the size of a maximum chain of constrained atoms, provides a tighter bound on the number of examples needed for pac-learning. To present this result, we need the following definition that applies the notion of *VC-dimension* [Blumer *et al.*, 1989] to constrained atoms.

Definition 21 *The* VC-dimension *of constrained atoms of head size at most n with respect to a constraint theory, Σ, is the size of the largest set S of such atoms for which the following holds. For any subset s of S, there exists a constrained atom that, with Σ, covers every constrained atom in s and covers no constrained atoms in $S - s$. Such a set S is said to be* shattered.

Theorem 22 *Let Σ be a constraint theory having C constraint predicates, and let the maximum of the arities of all constraint predicates in Σ be r. The VC-dimension, with respect to Σ, of constrained atoms having head size at most n is $Cn^r + n + 1$.*

A sketch of the proof of this result is in the appendix. Based on work in Computational Learning Theory[Blumer *et al.*, 1989], this result on the VC-dimension of constrained atoms implies that the number of examples needed for pac-learning is

$$\max\left(\frac{4}{\epsilon}\log\frac{2}{\delta}, \frac{8d}{\epsilon}\log\frac{13}{\epsilon}\right)$$

where d is Cn^r+n+1 and n is the head size of the target concept. Because C and r are constants for a fixed constraint theory, Σ, the VC-dimension is polynomial in $\frac{1}{\epsilon}$, $\frac{1}{\delta}$, and n.[9]

In general, the learnability of a class of concepts, such as constrained atoms, does not imply the learnability of a subclass, such as ordinary atoms or extended sorted atoms. In this case, however, the proof of the VC-dimension of constrained atoms applies to ordinary and extended sorted atoms as well. Therefore, the VC-dimension of ordinary atoms, for which the constraint theory is always empty, is $n+1$, and the VC-dimension of extended sorted atoms, for which the constraint theory can only contain monadic constraints, is $Cn+n+1$. The Ordinary Generalization and Extended SG Algorithms can be used to polynomially pac-learn ordinary atoms and extended sorted atoms (with respect to a fixed sort theory), respectively, in the same way the Constraint Generalization Algorithm is used to polynomially pac-learn constrained atoms.

5 Relationship to Other Work

Throughout most of the paper we have viewed sorted and constraint generalization as operations on atoms in constraint logic. We have noted that sorted and constrained atoms, universally closed, are equivalent to special kinds of definite clauses. This equivalence lets us relate these operations to the *generalized subsumption* of Buntine [1988]. Generalized subsumption is defined over definite clauses with respect to

[9] See [Anthony *et al.*, 1990] for another VC-dimension-based bound upper bound on the number of examples needed for pac-learning.

a logic program that acts as the background information. Because constrained atoms are equivalent to definite clauses whose antecedents, or bodies, contain only constraints on terms in the consequent, or head, we can compare Σ-subsumption, on which sorted and constraint generalization are based, with generalized subsumption, in cases where Σ consists entirely of definite clauses. In these cases, Σ-subsumption and generalized subsumption are equivalent. In the case where Σ-subsumption is based on information that comprises a constraint theory for which constraint queries are answered efficiently, the Constraint Generalization Algorithm can be used to efficiently generalize constrained atoms. Thus our work may be seen as identifying a restricted case for which Σ-subsumption is efficiently computable. An interesting area for further work is the search for other ways to extend the languages of atoms and background information while constraining it enough to preserve polynomial-time generalization based on Σ-subsumption.

Appendix

5.1 Proof of Theorem 20

First, we extend the definition of $\preceq_\Sigma$. As presently defined, $\tau_1 \preceq_\Sigma \tau_2$ if and only if constraint predicates τ_1 and τ_2 have the same arity, n, and $\Sigma \models \overline{\forall}(\tau_1(x_1, ..., x_n) \rightarrow \tau_2(x_1, ..., x_n))$. We now define $\preceq_\Sigma$ between *sets of constraint predicates of the same arity* in an analogous fashion: $\{\tau_1, ..., \tau_m\} \preceq_\Sigma \{\omega_1, ..., \omega_k\}$ if and only if $\tau_1, ..., \tau_m, \omega_1, ..., \omega_k$ all have the same arity, n, and $\Sigma \models \overline{\forall}(\tau_1(x_1, ..., x_n) \wedge ... \wedge \tau_m(x_1, ..., x_n) \rightarrow \omega_1(x_1, ..., x_n) \wedge ... \wedge \omega_k(x_1, ..., x_n))$. We also say in this case that the first set of constraints is *tighter than* the second. Note that for a constraint theory, Σ, containing C constraint predicates, the maximum chain of sets of constraint predicates ordered by $\preceq_\Sigma$ has length at most C. This is because any set in such a chain must contain some predicate that does not appear in a tighter set (a set below it) in the chain.

We now show that there can be no chain of more than $Cn^r + n + 2$ constrained atoms (ordered by $\geq_\Sigma$). We associate a *height* with each constrained atom α. Let m be the number of variables (not variable occurrences) in the head of α, and let k be the number of variable and function symbol occurrences in the head of α. For each tuple $T=\langle t_1, ..., t_j \rangle$ of $j \leq r$ terms in the head of α, if T is unconstrained then let $\#(T) = C$. Otherwise let N be the set of constraint predicates applied to T in the constraint of α; let $\#(T)$ be the maximum number of sets of constraint predicates that are below N in a chain of such sets. Let height(α) be

$$(n-k) + m + 1 + \sum_{\text{tuples } T \text{ of } j \leq r \text{ terms}} \#(T).$$

We first show that the size of a maximum chain of constrained atoms, each with head size no larger than n, is the maximum height of any constrained atom of head size at most n. This is shown by proving that the size of a maximum chain having greatest element α is at most height(α). The proof is by induction on height(α). An observation that is useful in the proof is that α is Σ-more general than another constrained atom, β, if and only if β results from tightening the set of constraint predicates on one or more tuples of terms in α or from substituting non-variable terms for one or more variables in α. This observation is straightforward given the definition of Σ-more general.

Basis: Height is 1. Then no sets of constraint predicates on tuples of terms in α can be tightened, and α contains no variables, so no substitutions can be applied. Therefore, α is at the bottom of every chain in which it appears.

Inductive: Tightening the set of constraint predicates on a tuple T of terms in α reduces $\#(T)$. Tightening a set of constraint predicates therefore reduces height(α). It remains to show that substituting a non-variable term for a variable, x, reduces the height. Such a substitution can only change m or k. Note that any non-variable term of length i can contain at most $i-1$ variables. Substituting such a term for x removes x from α and adds to α at most $i-1$ variables. Therefore, applying such a substitution increases m by at most $i-2$ and decreases k by $i-1$. Hence applying such a substitution reduces height(α).

To complete the proof, we show that the maximum value for height(α) is $Cn^r + n + 1$. Because there are at most n r-tuples of terms in α, and the length of the largest chain of constraints on each of these is C, the maximum value of

$$\sum_{\text{tuples } T \text{ of } j \leq r \text{ terms}} \#(T)$$

is Cn^r. Because m is at most k, the maximum value of $(n-k)+m+1$ is $n+1$. Therefore, the maximum value for height(α) is $Cn^r + n + 1$. □

[Notice that the size of a maximum chain does not increase if we allow the head, B, of the bottom constrained atom, β, to have head size greater than n but still require all others to have head size at most n. To see this, note that if B has head size greater than n, then it has head size greater than the size of the head (B') of the atom, β', above it. Then B' is not the same as B, but for some substitution θ, $B'\theta = B$. Then B' must have at least one variable, so the height of β' must be at least 2. But the height of the top element, α, in the chain can be at most Cn^r+n+1, so the chain from α to β', inclusive, has length at most $Cn^r + n$. This result is useful in the proof of Theorem 22.]

We note that for at least some constraint theories,

chains of length $Cn^r + n + 1$ do exist. Let Σ be a constraint theory whose constraint predicates are $\tau_1, ..., \tau_C$, each of arity r, such that $\tau_C \prec_\Sigma ... \prec_\Sigma \tau_1$. Let the greatest element in the chain be $p(x_1, ..., x_n)$. To get the each of the next n^r atoms of the chain, add to the constraint of the previous atom an atomic formula built from the predicate τ_1 and one of the n^r r-tuples of variables from $x_1, ..., x_n$ not used with τ_1 thus far. Repeat for each constraint $\tau_2, ..., \tau_C$ to build a chain of total length $Cn^r + 1$. Let the remaining n elements of the chain be the result of replacing, one at a time, the variables in the head with constants.

5.2 Proof Sketch for Theorem 22

We begin by showing that no set of $Cn^r + n + 2$ constrained atoms is shattered by constrained atoms of head size at most n. We do so by showing that if such a set were shattered, there would exist a chain of constrained atoms, all but the bottom of which have head size at most n, of length $Cn^r + n + 2$.[10] This would directly contradict the generalization of Theorem 20 given after the preceding proof. Let $\{\alpha_1, ..., \alpha_{Cn^r+n+2}\}$ be the set. Because the set is shattered, we can note that for all i between 1 and $Cn^r + n + 2$, inclusive, a constrained atom exists that covers $\alpha_1, ..., \alpha_i$ and covers no other members of the set. Then by the definition of msg_Σ, the msg_Σ, g_i, of α_1, ..., α_i is such a constrained atom. Then $g_{Cn^r+n+2} >_\Sigma ... >_\Sigma g_1$, yielding a chain of size $Cn^r + n + 2$.

We now give a simple example showing that for some constraint theories, a set of $Cn^r + n + 1$ constrained atoms can be shattered by constrained atoms whose heads have size at most n. Let Σ be the empty constraint theory. Then C is 0, so $Cn^r + n + 1 = n + 1$. It is enough to present a shattered set of size $n+1$ for a particular n. We present the set for $n = 4$, from which it is easy to also identify such sets for all other values of n. The set {*p(a,a,a,a), p(b,a,a,a), p(b,b,a,a), p(b,b,b,a), p(b,b,b,b)*} is shattered by constrained atoms of size at most 4. For each atom in the set, there is a way to exclude, or fail to cover, only that atom, and these ways of exclusion are orthogonal. For example, any atom that has a b for the first argument fails to cover the first atom. Any atom whose first two arguments are the same fails to cover the second atom. Any atom whose second and third arguments are the same fails to cover the third atom. Because these are orthogonal, they can be combined to exclude any subset of the atoms while including all others. Here are two examples of such combinations. The atom $p(x, y, y, y)$ excludes the third and fourth atoms because the second, third, and fourth arguments are the same. The atom $p(b, b, x, x)$ excludes the first atom because the first argument is a b, the second atom because the first and second arguments are the same, and the fourth atom because the third and fourth arguments are the same. For any subset we construct an atom that covers the atoms in the subset and excludes the others. Thus the set of five atoms is shattered by atoms of (head) size 4. □

[10] We thank Professor Douglas West of the University of Illinois Mathematics Department for a helpful conversation, that led us to consider this method of proof about VC-dimension within partially ordered sets.

References

[Angluin, 1988a] D. Angluin. *Negative Results for Equivalence Queries.* Technical Report YALEU/DCS/RR-648, Yale University, September 1988.

[Angluin, 1988b] D. Angluin. Queries and concept learning. *Machine Learning*, 2:319–342, 1988.

[Anthony *et al.*, 1990] Martin Anthony, Norman Biggs, and John Shawe-Taylor. The learnability of formal concepts. In *Proceedings of the 1990 Workshop on Computational Learning Theory*, pages 246–257, Morgan Kaufmann, Rochester, NY, August 1990.

[Blumer *et al.*, 1989] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth. Learnability and the vapnik-chervonenkis dimension. *JACM*, 36(4):929–965, 1989.

[Buntine, 1988] W. Buntine. Generalized subsumption and its applications to induction and redundancy. *Artificial Intelligence*, 36(2):149–176, 1988.

[Frisch and Page, 1990] A. M. Frisch and C. D. Page. Generalisation with taxonomic information. In *Proceedings of AAAI-90*, pages 755–761, Boston, MA, 1990.

[Lassez and Marriott, 1987] J-L. Lassez and K. Marriott. Explicit representation of terms defined by counter examples. *Journal of Automated Reasoning*, 3:301–317, 1987.

[Lassez *et al.*, 1988] J-L. Lassez, M. J. Maher, and K. Marriott. Unification revisited. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 15, pages 587–625, Morgan Kaufmann Publishers, 1988.

[Mitchell *et al.*, 1983] T. G. Mitchell, P. E. Utgoff, and R. Banerji. Learning by experimentation: Acquiring and redefining problem solving heuristics. In T.M. Mitchell R.S. Michalski, J.G. Carbonell, editor, *Machine Learning: An Artificial Intelligence Approach*, chapter 5, pages 137–162, Morgan Kaufmann Publishers, 1983.

[Pitt and Valiant, 1988] L. Pitt and L. G. Valiant. Computational limitations on learning from examples. *JACM*, 35(4):965–984, 1988.

[Plotkin, 1970] G. D. Plotkin. *A Note on Inductive Generalization*, chapter 8, pages 153–163. Volume 5 of *Machine Intelligence*, Edinburgh University Press, Edinburgh, 1970.

[Plotkin, 1971] G. D. Plotkin. *A Further Note on Inductive Generalization*, chapter 8, pages 101–124. Volume 6 of *Machine Intelligence*, Edinburgh University Press, 1971.

[Reynolds, 1970] J. C. Reynolds. *Transformational Systems and Algebraic Structure of Atomic Formulas*, chapter 7, pages 135–52. Volume 5 of *Machine Intelligence*, Edinburgh University Press, Edinburgh, 1970.

[Valiant, 1984] L. G. Valiant. A theory of the learnable. *Comm. ACM*, 27(11):1134–1142, 1984.

S11-61
93882
045143
DOP.

On the Power of Abstract Interpretation

Uday S. Reddy

Samuel N. Kamin

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana, IL 61801

Net: {reddy,kamin}@cs.uiuc.edu

September 17, 1991

Submitted for publication

Abstract

Increasingly sophisticated applications of static analysis place increased burden on the reliability of the analysis techniques. Often, the failure of the analysis technique to detect some information may mean that the time or space complexity of the generated code would be altered. Thus, it is important to precisely characterize the power of static analysis techniques.

We follow the approach of Sekar *et. al.* [13] who studied the power of strictness analysis techniques. Their result can be summarized by saying "strictness analysis is perfect up to variations in constants." In other words, strictness analysis is as good as it could be, short of actually distinguishing between concrete values. We use this approach to characterize a broad class of analysis techniques based on abstract interpretation including, but not limited to, strictness analysis.

- For the first-order case, we consider abstract interpretations where the abstract domain for data values is *totally ordered*. This condition is satisfied by Mycroft's strictness analysis, that of Sekar *et. al.* and Wadler's analysis of list-strictness. For such abstract interpretations, we show that the analysis is complete in the sense that, short of actually distinguishing between concrete values with the same abstraction, it gives the best possible information.
- We further generalize these results to typed lambda calculus with pairs and higher-order functions. Note that products and function spaces over totally ordered domains are *not* totally ordered. In fact, the notion of completeness used in the first-order case fails if product domains or function spaces are added. We formulate a weaker notion of completeness based on observability of values. Two values (including pairs and functions) are considered indistinguishable if their observable components are indistinguishable. We show that abstract interpretation of typed lambda calculus programs is complete upto this notion of indistinguishability.
- We use denotationally-oriented arguments instead of the detailed operational arguments used by Sekar *et. al.*. Hence, our proofs are much simpler. They should be useful for further future improvements.

1 Introduction

Static analysis allows compilers to optimize code generation. In recent times, static analysis based on the technique *abstract interpretation* has found increased use [2, 5, 10, 9]. In contrast to traditional applications of static analysis, the newer applications in functional and logic programming implementations achieve radical speedups, sometimes even altering the *complexity* of run time or space. Thus, these applications place an increased burden on the reliability of static analysis. The failure of the analysis to detect some information may mean that the complexity of the algorithm would be altered. It is therefore necessary to precisely characterize how powerful a static analysis technique is and to state it in terms that the user of a programming language can understand. The latter is necessary because the user of a sophisticated compiler would have to tune his program to fit the compiler's capabilities so that the desired performance can be achieved.

Clearly, no static analysis method can give precise information about all programs. If it did, that would be a violation of Rice's theorem that nontrivial semantics properties of computable functions are undecidable. So, analysis is in general approximate and it is hard to measure how good an approximation it is.

In a recent paper, Sekar *et. al.* [13] proposed a new approach to the problem. An abstraction map blurs certain distinctions in the semantic values. Hence, it can never find properties which depend on the distinctions being blurred. However, we may ask whether it finds all the properties which *do not* depend on such distinctions. If it can find all such properties, we can call it *complete*. Sekar *et. al.* apply this technique to two particular strictness analysis problems, *viz.*, Mycroft's strictness analysis method for first-order functions [9] and a method of their own formulation [14] which improves on Mycroft's analysis. They show that both these analyses are complete.

In this paper, we follow the approach of Sekar *et. al.* and obtain completeness results for a broad *class* of abstract interpretations. Our results show that whenever the base abstract domain is a finite total order, first-order abstract interpretation is complete in the sense just described. This subsumes the two completeness results proved by Sekar *et. al.* and it also covers other abstract interpretations such as the Wadler's analysis for "list strictness" [16]. Further, our proofs avoid the detailed operational arguments used by Sekar *et. al.* and focus on the abstract domain and semantic interpretation.

There are serious impediments to relaxing the condition that the base abstract domain must be a total order. For instance, if product domains are abstracted to products of the component abstract domains (which is not a total order) then abstract interpretation is *not* complete. Thus, some condition

is necessary even though the total order restriction may be too strong.

We also generalize the results for *typed higher-order* languages with product and function types. As indicated above, abstract interpretation for products (and function spaces) cannot be complete in the same sense as for the first-order case. So, we propose a revised notion of completeness where structured values such as pairs and functions are not treated as observable values, but have components that are be observable. Two structured values are considered indistinguishable (for the purposes of static analysis) if all their observable components are indistinguishable. This notion of indistinguishability leads to a new notion of completeness. We are able to show that, whenever abstract domains are constructed from totally ordered abstractions for observables, abstract interpretation is complete in that it gives the best information available for all indistinguishable programs.

1.1 Relation to previous work

Obviously, the closest work to ours is Sekar *et al.* [13]. We have already stated where we stand with respect to that work: by using denotational arguments instead of operational ones, our results are more general and our proofs simpler.

It is important to understand that this paper does not attempt to offer a new method of strictness analysis. Like Sekar *et al.*, we are only trying to pin down the power of the traditional techniques of Mycroft [9] and Burn, Hankin, and Abramsky [3].

A number of authors have discussed the optimality of static analyses (e.g. [1, 6, 11]). However, that work does not consider the same problem as concerns us. Where they consider optimality in general — i.e. finding the correct mathematical characterization of optimality — we are looking for a concrete characterization of the information loss in a particular abstract interpretation.

2 First order function definitions

2.1 Language

For simplicity, we assume that all functions are n-ary for some fixed $n \geq 0$. Let $x_1, \ldots, x_n$ be a set of n variables. The language includes a set of constants $k \in \mathit{Const}$, a set of primitive functions $p \in \mathit{Prim}$ and a set of function symbols $f \in \mathit{Fun}$. The abstract syntax is

$$\begin{array}{lcl} e & \in & \mathrm{Exp} \\ e & ::= & x_i \mid k \mid p(e_1, \ldots, e_n) \mid f(e_1, \ldots, e_n) \end{array}$$

A program is a set of equations of the form

$$f(x_1, \ldots, x_n) = e$$

for each f.

For the semantics, assume a cpo $\mathcal{D}$ for the value domain. Each constant symbols is interpreted as a value in $\mathcal{D}$. Each function symbol is interpreted as a continuous function in $[\mathcal{D}^n \to \mathcal{D}]$ and each primitive function symbol is assigned a fixed interpretation. We ambiguously denote the interpretation of a constant k by k, and the interpretation of a primitive function symbols p by p. Now, the semantics of expressions is determined with respect to an interpretation I mapping function symbols to functions in $[\mathcal{D}^n \to \mathcal{D}]$ and an environment η mapping variables to values in $\mathcal{D}$:

$$[\![\cdot]\!] \quad : \quad (Fun \to [\mathcal{D}^n \to \mathcal{D}]) \to (Var \to \mathcal{D}) \to D$$

$$\begin{aligned}
[\![x_i]\!]I\eta &= \eta(x_i) \\
[\![k]\!]I\eta &= k \\
[\![p(e_1, \ldots, e_n)]\!]I\eta &= p([\![e_1]\!]I\eta, \ldots, [\![e_n]\!]I\eta) \\
[\![f(e_1, \ldots, e_n)]\!]I\eta &= I(f)([\![e_1]\!]I\eta, \ldots, [\![e_n]\!]I\eta)
\end{aligned}$$

An interpretation I that satisfies all the equations of the program is a *model* of the program, and the standard semantics of the program is its least model. As usual, the least model can be expressed as the least fixed point of a functional F on interpretations. Let $F(I)$ be the interpretation I' such that, for all equations $f(\bar{x}) = e$ in the program,

$$[\![f(\bar{x})]\!]I'\eta = [\![e]\!]I\eta$$

Models of the program are precisely the fixed points of F. Clearly, F is continuous and, so, the least fixed point is given by the lub of the Kleene iteration sequence.

2.2 Abstract semantics

For the abstract interpretation, assume a complete lattice $\mathcal{A}$. There must be an abstraction function $abs : \mathcal{D} \to \mathcal{A}$ that is continuous. The abstraction function induces a corresponding function $abs : [\mathcal{D}^n \to \mathcal{D}] \to [\mathcal{A}^n \to \mathcal{A}]$ defined by

$$abs(f)(\bar{a}) = \sqcup\{abs(f(\bar{v})) \mid abs(\bar{v}) \sqsubseteq \bar{a}\}$$

The abstraction of constant and primitive symbols are $k^{\mathrm{I}} = abs(k)$ and $p^{\mathrm{I}} = abs(p)$ respectively. A program can be given semantics in the abstract domain by defining a semantic function $[\![e]\!]^{\mathrm{I}}$. Again,

the least abstract model is given by the least fixed point of a continuous function F on abstract interpretations. The following results relate the standard semantics and the abstract semantics. Whenever I is an interpretation, $I^\sharp$ denotes the abstract interpretation given by $I^\sharp(f) = abs(I(f))$. Similarly, if η is an environment, $\eta^\sharp$ denotes the corresponding abstract environment $\eta^\sharp(x) = abs(\eta(x))$.

Theorem 2.1 (Soundness for expressions) $abs([\![e]\!]I\eta) \sqsubseteq_{\mathcal{A}} [\![e]\!]^\sharp I^\sharp \eta^\sharp$.

Theorem 2.2 (Soundness for programs) If I is the standard semantics of a program and I_A is its least abstract semantics, then $I^\sharp \sqsubseteq_{\mathcal{A}} I_A$.

For the purpose of this paper, we impose two conditions on the abstractions. First, $\mathcal{A}$ must be a finite *linearly ordered* partial order. (Such a partial order is automatically a complete lattice). Second, the abstraction map $abs : \mathcal{D} \to \mathcal{A}$ should be *upward-faithful*, i.e., whenever $abs(v) \sqsubseteq_{\mathcal{A}} abs(w)$, there exists a $w' \in \mathcal{D}$ such that $abs(w') = abs(w)$ and $v \sqsubseteq_{\mathcal{D}} w'$. This essentially means that $a \sqsubseteq_{\mathcal{A}} b$ iff the inverse images of a and b are related by the Hoare power domain order. See [3] for a discussion of the use of Hoare power domains in abstract interpretation. of other elements.

With these conditions, we can define a canonical subset of $\mathcal{D}$ that will be useful in future proofs. Specifically, we can select a sequence $C = c_0 \sqsubseteq c_1 \sqsubseteq \ldots \sqsubseteq c_n \sqsubseteq \mathcal{D}$ as follows: $c_0 = \bot$, and for every $a \sqsupseteq abs(\bot)$, there is a unique i such that $abs(c_i) = a$. Then for any function $f \in \mathcal{D} \to \mathcal{D}$, define $f|_C$ by:

$$f|_C(x) \;=\; \text{the } c_i \text{ such that } abs(c_i) = abs(f(x))$$

$|_C$ is a kind of normalizer for functions. It is useful for proving the following lemma:

Lemma 2.3 If $f \in \mathcal{D} \to \mathcal{D}$ is an arbitrary (possibly non-monotonic) function with $abs \circ f$ monotonic, then there exists a monotonic f' with $abs \circ f' = abs \circ f$.

Proof: Let $f' = f|_C$.

$|_C$ is also used in the proof of the first part of the following lemma, which shows that upward-faithfulness generalizes to functions, environments and interpretations in a natural way:

Lemma 2.4 Let $abs : \mathcal{D} \to \mathcal{A}$ be an upward-faithful abstraction function.

1. For $f, g \in \mathcal{D}^n \to \mathcal{D}$, if $abs(f) \sqsubseteq abs(g)$ then there exists g' such that $abs(g') = abs(g)$ and $f \sqsubseteq g'$.
2. For environments η_1 and η_2, if $\eta_1^\sharp \sqsubseteq \eta_2^\sharp$ then there exists η_2' such that $(\eta_2')^\sharp = \eta_2^\sharp$ and $\eta_1 \sqsubseteq \eta_2$.
3. For interpretations I and J, if $I^\sharp \sqsubseteq J^\sharp$ then there exists J' such that $(J')^\sharp = J^\sharp$ and $I \sqsubseteq J'$.

The conditions we have imposed on $\mathcal{A}$ allow us to simplify the definition of *abs* for functions:

Lemma 2.5 For any $f \in \mathcal{D}^n \to \mathcal{D}$ and $\bar{a} \in \mathcal{A}^n$ such that all the components of $\bar{a}$ are in the range of *abs*,

$$abs(f)(\bar{a}) = \sqcup\{abs(f(\bar{v})) \mid abs(\bar{v}) = \bar{a}\}.$$

Proof: Let $\bar{w}$ be such that $abs(\bar{w}) = \bar{a}$. By definition,

$$abs(f)(\bar{a}) = abs(f)(abs(\bar{w})) = \sqcup\{abs(f(\bar{u})) \mid abs(\bar{u}) \sqsubseteq abs(\bar{w})\}.$$

Upward-faithfulness implies that this equals:

$$\sqcup\{abs(f(\bar{u})) \mid \exists \bar{v}.abs(\bar{v}) = abs(\bar{w}) \wedge \bar{u} \sqsubseteq \bar{v}\},$$

which by monotonicity of *abs* equals:

$$\sqcup\{abs(f(\bar{v})) \mid abs(\bar{v}) = abs(\bar{w})\},$$

as required.

Lemma 2.6 For all f and $\bar{v}$, there exists a $\bar{u}$ such that $abs(\bar{v}) = abs(\bar{u})$ and $abs(f)(abs(\bar{v})) = abs(f(\bar{u}))$.

Proof: This follows from the previous lemma, together with the fact that $\mathcal{A}$ is finite and linearly-ordered, which implies that $\sqcup$ is just the max operation.

2.3 Similarity

The abstraction map induces an equivalence relation $\sim$ (called *similarity*) on $\mathcal{D}$ and $[\mathcal{D}^n \to \mathcal{D}]$ as follows. For $v, w \in \mathcal{D}$,

$$v \sim w \Leftrightarrow abs(v) = abs(w)$$

For $f, g \in [\mathcal{D}^n \to \mathcal{D}]$,

$$f \sim g \Leftrightarrow \forall \bar{v} \in \mathcal{D}^n.\ f(\bar{v}) \sim g(\bar{v})$$

The idea of similarity is that the abstract interpretation has no way to distinguish between values and functions that are similar. However, the similarity of functions is slightly weaker than having the same abstraction. The notion of similarity can be extended to environments and interpretations point-wise.

Example Let $Bool_\perp$ be the flat cpo of truth values, and $2 = \{\perp, \top\}$ be the "definedness" abstract domain with the abstraction function $abs = \{\perp \mapsto \perp, true \mapsto \top, false \mapsto \top\}$. The only similarity in $Bool_\perp$ is $true \sim false$. Some of the functions in $[Bool_\perp \to Bool_\perp]$ are

	$\perp$	*true*	*false*
abort	$\perp$	$\perp$	$\perp$
succ	$\perp$	*true*	$\perp$
nsucc	$\perp$	*false*	$\perp$
fail	$\perp$	$\perp$	*false*
nfail	$\perp$	$\perp$	*true*
id	$\perp$	*true*	*false*
not	$\perp$	*false*	*true*

The function *abort* is not similar to any other function. Among the others, $succ \sim nsucc$, $fail \sim nfail$ and $id \sim not$. Note that all these latter functions have the same abstraction. So, similarity is weaker than having the same abstraction. In Section 3, we strengthen the definition of similarity of functions to include $succ \sim fail$. The idea here is that each of them can simulate the other by modifying its input to a similar value. However, $succ \sim id$ does not hold even for the strengthened definition.

Abstract interpretation of expressions looses information, in general. For instance, whenever $x \neq \perp$, $succ(fail(x)) = \perp$, but $succ^!(fail^!(x)) = \top$. However, such a loss of information *can* be justified. Since abstract interpretation cannot distinguish between similar values and similar functions, it can only give information that is shared by all the expressions similar to the given expression. Since $fail \sim nfail$ and $succ(nfail(x)) = true$, we cannot expect the abstract interpretation to give precise information about $succ(fail(x))$. □

2.4 Similarity semantics

We formalize the loss of information as follows. We associate with an expression a *class* of interpretations each of which is obtained by confusing the information in the expression via similarity. Now, we can say that an abstraction is *complete* if it identifies the properties shared by all interpretations in the class.

Define a relation $\rightsquigarrow_{I,\eta} \subseteq (Exp \times \mathcal{D})$ which relates expressions to values by confusing the values of

subexpressions:

$$\begin{array}{rcll} x & \leadsto_{I,\eta} & v & \text{if } \eta(x) \sim v \\ k & \leadsto_{I,\eta} & k' & \text{if } k \sim k' \\ p(e_1,\ldots,e_n) & \leadsto_{I,\eta} & p'(v_1,\ldots,v_n) & \text{if } p \sim p' \text{ and } e_i \leadsto_{I,\eta} v_i \\ f(e_1,\ldots,e_n) & \leadsto_{I,\eta} & f'(v_1,\ldots,v_n) & \text{if } I(f) \sim f' \text{ and } e_i \leadsto_{I,\eta} v_i \end{array}$$

Note that the left hand sides of these rules are *expressions* and the right hand sides are *semantic values*. The $\leadsto$ relation is similar to the semantic function except that it allows the values of subexpressions to be modified up to similar values. This gives the following results:

Lemma 2.7 $e \leadsto_{I,\eta} [\![e]\!] I \eta$.

Secondly, $\leadsto$ is indifferent to similarity perturbations in the inputs and results.

Lemma 2.8 If $e \leadsto_{I,\eta} v$, $I \sim I'$, $\eta \sim \eta'$ and $v \sim v'$ then $e \leadsto_{I',\eta'} v'$.

Now, the following result strengthens Theorem 2.1 by replacing $\sqsubseteq_{\mathcal{A}}$ with equality.

Theorem 2.9 (Completeness for expressions) Let I be an interpretation and η an environment. For all expressions e, there is a $v \in \mathcal{D}$ such that $e \leadsto_{I,\eta} v$ and $abs(v) = [\![e]\!]^{\mathbf{I}} I^{\mathbf{I}} \eta^{\mathbf{I}}$.

Proof: By structural induction on e.

- $[\![x]\!]^{\mathbf{I}} I^{\mathbf{I}} \eta^{\mathbf{I}} = \eta^{\mathbf{I}}(x) = abs(\eta(x))$. Since $x \leadsto_{I,\eta} \eta(x)$, the conclusion is obtained.
- $[\![k]\!]^{\mathbf{I}} I^{\mathbf{I}} \eta^{\mathbf{I}} = k^{\mathbf{I}} = abs(k)$. Since $k \leadsto_{I,\eta} k$, the conclusion is obtained.
- $[\![p(\bar{e})]\!]^{\mathbf{I}} I^{\mathbf{I}} \eta^{\mathbf{I}} = p^{\mathbf{I}}([\![\bar{e}]\!]^{\mathbf{I}} I^{\mathbf{I}} \eta^{\mathbf{I}})$. By inductive hypothesis, there exist $\bar{v}$ such that $\bar{e} \leadsto_{I,\eta} \bar{v}$ and $abs(\bar{v}) = [\![\bar{e}]\!]^{\mathbf{I}} I^{\mathbf{I}} \eta^{\mathbf{I}}$. Now, $p^{\mathbf{I}}(abs(\bar{v})) = abs(p(\bar{u}))$ for some $\bar{u} \sim \bar{v}$, by lemma 2.6. Since $\bar{e} \leadsto_{I,\eta} \bar{v}$ and $\bar{v} \sim \bar{u}$, we also have $\bar{e} \leadsto_{I,\eta} \bar{u}$. Hence, the conclusion follows.
- $[\![f(\bar{e})]\!]^{\mathbf{I}} I^{\mathbf{I}} \eta^{\mathbf{I}} = I^{\mathbf{I}}(f)([\![e]\!]^{\mathbf{I}} I^{\mathbf{I}} \eta^{\mathbf{I}})$. This case is similar to the above because $I^{\mathbf{I}}(f) = abs(I(f))$.

□

This result is somewhat stronger than what is needed. It says that, among the interpretations associated with an expression, there is one that precisely realizes the information given by the abstract interpretation. Hence, it follows that the abstract interpretation gives the information shared by all the interpretations of the expression.

An interpretation I is said to be a *similarity-model* of a program if, for each equation $f(\bar{x}) = e$ in the program,

$$e \rightsquigarrow_{I,\eta} [\![f(\bar{x})]\!] I\eta$$

However, unlike for the standard semantics, similarity-models cannot be expressed as fixed points. We use an alternative formulation to achieve the same effect. Given two interpretations I and J of a program P, we say that I improves to J ($I \rightsquigarrow_P J$) iff, for every equation $f(\bar{x}) = e$,

$$e \rightsquigarrow_{I,\eta} [\![f(\bar{x})]\!] J\eta$$

Like the $\rightsquigarrow_{I,\eta}$ relation, $\rightsquigarrow_P$ is indifferent to similarity perturbations:

Lemma 2.10 If $I' \sim I$, $I \rightsquigarrow_P J$ and $J \sim J'$ then $I' \rightsquigarrow_P J'$.

Proof: The only use made of I in $I \rightsquigarrow_P J$ is in the condition $e \rightsquigarrow_{I,\eta} [\![f(\bar{x})]\!] J\eta$ which is indifferent to a similarity perturbation in I. Similarly, the only use made of J is in interpreting $f(\bar{x})$. Since $[\![f(\bar{x})]\!] J\eta = J(f)(\eta(\bar{x})) \sim J'(f)(\eta(\bar{x}))$, we have that $I' \rightsquigarrow_P J'$.[1] □

Note that I is a similarity-model of P iff $I \rightsquigarrow_P I$. Further:

Lemma 2.11 Every model of P is a similarity-model of P.

Proof: If I is a model of P then, for every equation $f(\bar{x}) = e$, $[\![f(\bar{x})]\!] I\eta = [\![e]\!] I\eta$. By Lemma 2.7, we have $e \rightsquigarrow_{I,\eta} [\![f(\bar{x})]\!] I\eta$. Hence, $I \rightsquigarrow_P I$. □

Intuitively, a similarity-model allows constants and primitives to be "confused". As a trivial special case, it may perform no confusion at all, in which case it is a model.

The predicate $e \rightsquigarrow_{I,\eta} v$ is inclusive in I, η and v, *i.e.*, given increasing sequences $\{I_i\}_i$ $\{\eta_i\}_i$ and $\{v_i\}_i$ such that $e \rightsquigarrow_{I_i,\eta_i} v_i$, we have $e \rightsquigarrow_{\sqcup_i I_i, \sqcup_i \eta_i} \sqcup_i v_i$. This is clear from the fact that the values v have a continuous dependence on η and I. Similarly, the relation $\rightsquigarrow_P$ is also inclusive. We can now give a Kleene-like construction for similarity-models:

Theorem 2.12 Let P be a program and $I_\perp = I_0 \rightsquigarrow_P I_1 \rightsquigarrow_P I_2 \rightsquigarrow_P \ldots$ be an increasing sequence of interpretations. $\sqcup_i I_i$ is a similarity-model of P.

[1] Even though $[\![f(\bar{x})]\!] J\eta \sim [\![f(\bar{x})]\!] J'\eta$, it is important to note that this does not generalize to arbitrary expressions in place of $f(\bar{x})$. For example, $f(g(x))$ has dissimilar values in the two similar interpretations $J = \{f \mapsto succ, g \mapsto fail\}$ and $J' = \{f \mapsto succ, g \mapsto nfail\}$.

Proof: Consider the subsequence $I_1 \rightsquigarrow_P I_2 \rightsquigarrow_P \ldots$. The given sequence is related to this by $\rightsquigarrow_P$ relation. Hence, by inclusivity, $\sqcup_{i\geq 0} I_i \rightsquigarrow_P \sqcup_{i\geq 1} I_i$. But, the two lubs are equal. Hence $\sqcup_i I_i \rightsquigarrow_P \sqcup_i I_i$ showing that it is a similarity-model. □

We call similarity-models expressible as lubs of such sequences *reachable similarity-models.* They capture the notion of confusing the standard semantics of programs. Other unreachable similarity-models are of no computational interest.

Theorem 2.13 (Completeness for programs) If J_A is the least abstract interpretation of a program P, there is a reachable similarity-model I of P such that $I^\natural = J_A$.

Proof: Let $F^\natural$ be the program's functional on abstract interpretations, i.e., for each equation $f(\bar{x}) = e$, $F^\natural(J)(f)(\bar{w}) = [\![e]\!]^\natural J\{\bar{x} \mapsto \bar{w}\}$. J_A is the least fixed point of $F^\natural$. Hence, it is the lub of the increasing sequence $J_i = (F^\natural)^i(J_\perp)$. We exhibit an increasing sequence $I_0 \rightsquigarrow_P I_1 \rightsquigarrow_P \ldots$ such that $(I_i)^\natural = J_i$.

- Let I_0 be the undefined interpretation. Clearly, $(I_0)^\natural = J_0$.
- Suppose $(I_i)^\natural = J_i$ for an i. For every expression e and every environment η, there is a $v \in \mathcal{D}$ such that $e \rightsquigarrow_{I_i,\eta} v$ and $abs(v) = [\![e]\!]^\natural (I_i)^\natural \eta^\natural$. Let f be a function symbol defined by $f(\bar{x}) = e$ and η the environment $\{\bar{x} \mapsto \bar{w}\}$ for some $\bar{w} \in \mathcal{D}^n$. We can define $I_{i+1}(f)(\bar{w})$ to be the value v such that $e \rightsquigarrow_{I_i,\eta} v$ and $abs(v) = [\![e]\!]^\natural (I_i)^\natural \eta^\natural$. We have $I_i \rightsquigarrow_P I_{i+1}$ from the condition $e \rightsquigarrow_{I_i,\eta} v$, and $(I_{i+1})^\natural = J_{i+1}$ from the condition $abs(v) = [\![e]\!]^\natural J_i \eta^\natural$.

  Recall now our canonical sequence $C \subseteq \mathcal{D}$ and function normalizer $|_C$. We have exactly the situation described in lemma 2.3: I_{i+1} is not necessarily monotonic as defined, but $abs \circ I_{i+1}$ is, so we can replace I_{i+1} by $I' = I_{i+1}|_C$ to ensure monotonicity. Moreover, if we assume the range of I_i is contained in C, then $abs \circ I_i \sqsubseteq abs \circ I'$ implies $I_i \sqsubseteq I'$.

Since $(I_i)^\natural = J_i$ for all i, $(\sqcup_i I_i)^\natural = J_A$. □

This result strengthens Theorem 2.2 by showing the existence of an interpretation I such that $I^\natural$ is equal to the abstract semantics. The interpretation I is obtained by confusing the standard semantics of the program. Since all such confusions are indistinguishable in the abstraction, this is indeed the best that can be achieved by abstract interpretation.

Note that the sequence J_i actually converges in finite time, say at element J_n, so that in fact $I_n^\# = J_A$. This suggests the following:

Corollary 2.14 If J_A is the least abstract interpretation of program P, there is a program P' obtained from P by a finite number of unfoldings of function definitions, followed by replacements of constants by similar constants, such that $[\![P]\!]^{\natural} = J_A$.

2.5 Discussion

To get a flavor of the completeness result, consider the program

$$f(x) = \textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } \bot$$

Given the abstraction $abs : Nat_\bot \to \mathbf{2}$ which maps $\bot_{Nat}$ to $\bot$ and all other values to $\top$, the abstraction of f is

$$f^{\natural}(x) = \begin{cases} \bot, \text{if } x = \bot \\ \top, \text{otherwise} \end{cases}$$

There is loss of information involved in this abstraction because, for instance, $f(1) = \bot$ which abstracts to $\bot$ whereas $f^{\natural}(abs(1)) = \top$. However, the result $f(1) = \bot$ depends on the fact that $(1 = 0) = \mathit{false}$. Since the abstraction map has been defined to blur the distinction between true and false, the abstract interpretation of $f(1)$ does not have the information that $(1 = 0) = \mathit{false}$. It would have to produce the same results even if $(1 = 0)$ were true. Our similarity semantics of programs is defined in such a way that it blurs all the distinctions that are lost in the abstraction. Both $1 = 0 \rightsquigarrow \mathit{false}$ and $1 = 0 \rightsquigarrow \mathit{true}$ are possible. Hence, both $f(1) \sim_{I,\eta} \bot$ as well as $f(1) \sim_{I,\eta} 1$ hold and, so, the result of abstract interpretation denotes the best possible information.

The main condition used in this completeness result is that the abstract domain must be linearly ordered. This may appear to be a severe restriction, but some of the oft-used abstract interpretations are based on linearly ordered abstract domains. The strictness analysis of Mycroft [10, 9] uses the two point definedness domain $\mathbf{2} = \{\bot \sqsubseteq \top\}$. Sekar *et. al.* [13] work with concrete domains of terms and analyze for three properties: $\bot$ denoting undefined, H denoting definedness of the head (outermost constructor symbol), and $\top$ denoting complete definedness. This analysis can be modeled by the abstract domain $\{\bot \sqsubseteq H \sqsubseteq \top\}$ which is linearly ordered. Wadler [16] uses a four-point abstract domain for analyzing list strictness which is also linearly ordered. Our results show that all these analyses are complete in that they provide the best information representable in these abstract domains.

It seems that the linear order restriction can be relaxed. What is needed is some kind of condition stating that the lubs of abstract values are "precise". That is, when we take the lub of two abstract

values for lack of certainty about which of two computation paths would be followed, the lub should precisely capture this uncertainty without introducing any extra loss of information. However, we are not able to formalize this intuition.

To give a flavor of loss of completeness that arises with imprecise lubs, consider the concrete domain $Nat_\perp \times Nat_\perp$ abstracted to 2×2. The pair $(\top, \top)$ is the lub of $(\perp, \top)$ and $(\top, \perp)$. Consider the program

$$f(x) = \textbf{if } x = 0 \textbf{ then } (1, \perp) \textbf{ else } (\perp, 2)$$

Its abstraction is

$$f^!(x) = \begin{cases} (\perp, \perp) & \text{if } x = \perp \\ (\top, \top) & \text{otherwise} \end{cases}$$

If $x \neq \perp$, $f(x)$ is either $(1, \perp)$ or $(\perp, 2)$ neither of which realizes the analysis $(\top, \top)$. More precisely, there is no v such that $f(x) \rightsquigarrow_{I,\eta} v$ and $abs(v) = (\top, \top)$. Theorem 2.9 fails. Thus, analysis of product domains cannot be complete in the sense of this section. In the next section, we adopt these notions to typed lambda calculus. It would then be seen that analysis of product domains can be complete in a different sense.

3 Typed lambda calculus

In this section, we extend the results of Section 2 to typed lambda calculus with products and function spaces. The language includes a set of base types β and a set of constants k^τ (which subsume primitive functions). The abstract syntax is

$$\begin{array}{rcl} \tau & \in & \textbf{Type} \\ \tau & ::= & \beta \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2 \\ e^\tau & \in & \textbf{Exp}_\tau \\ e^\tau & ::= & x^\tau \mid k^\tau \mid e^{\tau \times \tau'}.1 \mid e^{\tau' \times \tau}.2 \mid e_1^{\tau' \rightarrow \tau} e_2^{\tau'} \mid fix\, e^{\tau \rightarrow \tau} \\ e^{\tau_1 \times \tau_2} & ::= & \langle e_1^{\tau_1}, e_2^{\tau_2} \rangle \\ e^{\tau \rightarrow \tau'} & ::= & \lambda x^\tau . e^{\tau'} \end{array}$$

To be precise, one would have to define the context-sensitive syntax in terms of type inference rules as in [4, 8]. We often drop type superscripts because they can be automatically reconstructed.

For the semantics, assume a cpo $\mathcal{D}_\beta$ for each base type β. $\mathcal{D}_{\tau_1 \times \tau_2}$ is the cpo-product $\mathcal{D}_{\tau_1} \times \mathcal{D}_{\tau_2}$ and $\mathcal{D}_{\tau_1 \rightarrow \tau_2}$ is the continuous function space $[\mathcal{D}_{\tau_1} \rightarrow \mathcal{D}_{\tau_2}]$. The assignment $\mathcal{D}$ of domains to types is called

a *typed lambda structure*. The semantic functions are conventional:

$$\begin{aligned}
[\![x]\!]\eta &= \eta(x) \\
[\![k]\!]\eta &= k \\
[\![\langle e_1, e_2\rangle]\!]\eta &= \langle [\![e_1]\!]\eta, [\![e_2]\!]\eta\rangle \\
[\![e.1]\!]\eta &= fst([\![e]\!]\eta) \\
[\![e.2]\!]\eta &= snd([\![e]\!]\eta) \\
[\![\lambda x.\, e]\!]\eta &= \lambda v.\, [\![e]\!]\eta[x{\rightarrow}v] \\
[\![e_1\ e_2]\!]\eta &= ([\![e_1]\!]\eta)\,([\![e_2]\!]\eta) \\
[\![fix\ e]\!]\eta &= fix\,([\![e]\!]\eta)
\end{aligned}$$

For the abstract semantics, choose a similar typed lambda structure $\mathcal{A}$ with the restriction that each $\mathcal{A}_\beta$ is linearly ordered. The abstraction function $abs_\beta : \mathcal{D}_\beta \rightarrow \mathcal{A}_\beta$ should be continuous and upward-faithful. It is extended to other types by:

$$\begin{aligned}
abs_{\tau_1\times\tau_2}(\langle v_1, v_2\rangle) &= \langle abs_{\tau_1}(v_1), abs_{\tau_2}(v_2)\rangle \\
abs_{\tau_1\rightarrow\tau_2}(f) &= \lambda a \in \mathcal{A}_{\tau_1}.\ \sqcup\{abs_{\tau_2}(f(v)) \mid abs(v) \sqsubseteq_{\mathcal{A}} a\}
\end{aligned}$$

Note that abs_τ is continuous for each τ. The abstract semantic function is denoted by $[\![e]\!]^{\mathbf{I}}$.

Theorem 2.1 relating the standard and abstract semantics generalizes to the typed lambda calculus [3].

Theorem 3.1 (Soundness) $abs_\tau([\![e^\tau]\!]\eta) \sqsubseteq [\![e^\tau]\!]^{\mathbf{I}}\eta^{\mathbf{I}}$.

This subsumes Theorem 2.2 by the presence of *fix* terms.

The notion of similarity is now extended to typed lambda structures by defining $\sim_\tau$ for each type τ as follows:

- $v \sim_\beta w \Leftrightarrow abs_\beta(v) = abs_\beta(w)$.
- $p \sim_{\tau_1\times\tau_2} q \Leftrightarrow fst(p) \sim_{\tau_1} fst(q) \wedge snd(p) \sim_{\tau_2} snd(q)$.
- $f \sim_{\tau_1\rightarrow\tau_2} g \Leftrightarrow \forall v \in \mathcal{D}_{\tau_1}.\ (\exists w \sim_{\tau_1} v.\ f(v) \sim_{\tau_2} g(w)) \wedge (\exists w \sim_{\tau_1} v.\ g(v) \sim_{\tau_2} f(w))$.

In contrast to the treatment in Section 2.5, pairs are treated here as unobservable structures. So, two pairs are similar iff all their observable components are similar. Two functions are similar if, for every input, there is a similar input so that the two functions produce similar results for the two similar inputs. For example, among the functions in $[Bool_\perp \rightarrow Bool_\perp]$ mentioned in section 2.3, $succ \sim nsucc \sim fail \sim$

$nfail$ and $id \sim not$. So, this notion of similarity is stronger than that used in section 2 for the first-order case. However, the results of that section hold for the stronger notion of similarity as well because similarity interpretations of terms allow similarity perturbations in function inputs.

Define a relation $\rightsquigarrow_{\tau,\eta} \subseteq (Exp_\tau \times \mathcal{D}_\tau)$ for recursion-free terms as follows:

$$
\begin{array}{rlll}
x & \rightsquigarrow_{\tau,\eta} & v & \text{if } \eta(x) \sim v \\
k & \rightsquigarrow_{\tau,\eta} & k' & \text{if } k \sim k' \\
\langle e_1, e_2 \rangle & \rightsquigarrow_{\tau_1 \times \tau_2,\eta} & \langle v_1, v_2 \rangle & \text{if } e_1 \rightsquigarrow_{\tau_1,\eta} v_1 \text{ and } e_2 \rightsquigarrow_{\tau_2,\eta} v_2 \\
e.1 & \rightsquigarrow_{\tau_1,\eta} & fst(p) & \text{if } e \rightsquigarrow_{\tau_1 \times \tau_2,\eta} p \\
e.2 & \rightsquigarrow_{\tau_2,\eta} & snd(p) & \text{if } e \rightsquigarrow_{\tau_1 \times \tau_2,\eta} p \\
\lambda x.\, e & \rightsquigarrow_{\tau_1 \to \tau_2,\eta} & f & \text{if } \forall v.\ e \rightsquigarrow_{\tau_2,\eta[x \to v]} f(v) \\
e_1\, e_2 & \rightsquigarrow_{\tau_2,\eta} & f(v) & \text{if } e_1 \rightsquigarrow_{\tau_1 \to \tau_2,\eta} f \text{ and } e_2 \rightsquigarrow_{\tau_1,\eta} v
\end{array}
$$

Further, for all expressions e of appropriate types:

$$
\begin{array}{rlll}
e & \rightsquigarrow_{\tau_1 \times \tau_2,\eta} & \langle v_1, v_2 \rangle & \text{if } e.1 \rightsquigarrow_{\tau_1,\eta} v_1 \text{ and } e.2 \rightsquigarrow_{\tau_2,\eta} v_2 \\
e & \rightsquigarrow_{\tau_1 \to \tau_2,\eta} & f & \text{if for all } e',\ e' \rightsquigarrow_{\tau_1,\eta} v \Rightarrow ee' \rightsquigarrow_{\tau_1,\eta} f(v)
\end{array}
$$

The last two conditions denote the extra loss of information for structured values.

For recursive terms, we need some definitions. Let $F \subseteq [\mathcal{D}_\tau \to \mathcal{D}_\tau]$ be a similarity-closed set of functions, i.e., $f \in F$ and $f' \sim f$ implies $f' \in F$. Define the improvement relation $\rightsquigarrow_F \subseteq (\mathcal{D}_\tau \times \mathcal{D}_\tau)$ as $v \rightsquigarrow_F w$ iff $w = f(v)$ for some $f \in F$. A ***similarity-fixpoint*** of F is a value v such that $v \rightsquigarrow_F v$.

Lemma 3.2 $\rightsquigarrow_F$ is inclusive, i.e., whenever $\{v_i\}$ and $\{w_i\}$ are increasing sequences such that $v_i \rightsquigarrow_F w_i$, $\sqcup_i v_i \rightsquigarrow_F \sqcup_i w_i$.

Proof: Since the abstract domains are finite, there exists a finite k such that $v_k \sim v_i$ and $w_k \sim w_i$ for all $i \geq k$. Let $f_k \in F$ be the function such that $w_k = f_k(v_k)$. Since similarity is closed under lubs of sequences (*abs* is continuous), $v_k \sim \sqcup_i v_i$ and $w_k \sim \sqcup_i w_i$. Hence, there is a function $f \in F$ such that $\sqcup_i w_i = f(\sqcup_i v_i)$. ■

Theorem 3.3 Let $\perp_\tau = v_0 \rightsquigarrow_F v_1 \rightsquigarrow_F \ldots$ be an increasing improvement sequence. Then $\sqcup_i v_i$ is a similarity-fixpoint of F.

Proof: Consider the subsequence $v_1 \rightsquigarrow_f v_2 \rightsquigarrow_F \ldots$. The given sequence is related to this by the $\rightsquigarrow_F$ relation. Since $\rightsquigarrow_F$ is inclusive, $\sqcup_{i \geq 0} v_i \rightsquigarrow_F \sqcup_{i \geq 1} v_i$. The two lubs being equal, we have $\sqcup_i v_i \rightsquigarrow_F \sqcup_i v_i$. ■

As in the first-order case, we call the similarity-fixpoints expressible by lubs of such sequences *reachable*. Now, the similarity semantics of $fix\ e$ is specified as follows:

$$fix\ e \quad \leadsto_{\tau,\eta} \quad v \quad \text{if } v \text{ is a reachable similarity-fixpoint of } \{f \mid e \leadsto_{\tau\to\tau,\eta} f\}$$

The basic properties of the $\leadsto$ relation generalize from the first-order case:

Lemma 3.4 $e \leadsto_\eta [\![e]\!]\eta$.

Lemma 3.5 If $e \leadsto_\eta v$, $\eta \sim \eta'$ and $v \sim v'$ then $e \leadsto_{\eta'} v'$.

Lemma 3.6 The relation $e \leadsto_\eta v$ is inclusive in η and v, i.e., if $\{\eta_i\}$ and $\{v_i\}$ are increasing sequences of environments and values respectively such that $e \leadsto_{\eta_i} v_i$ then $e \leadsto_{\sqcup_i \eta_i} \sqcup_i v_i$.

Proof: Denote the lubs of $\{\eta_i\}$ and $\{v_i\}$ by η^∞ and v^∞ respectively. The proof is by structural induction on e. We show selected cases:

- If $x \leadsto_{\eta_i} v_i$ then $v_i \sim \eta_i(x)$. By inductive hypothesis, $v^\infty \sim (\eta^\infty)(x)$. Hence, the conclusion follows.
- If $\langle e, e' \rangle \leadsto_{\eta_i} \langle v_i, w_i \rangle$ then $e \leadsto_{\eta_i} v_i$ and $e' \leadsto_{\eta_i} w_i$. By inductive hypothesis, $e \leadsto_{\eta^\infty} v^\infty$ and $e' \leadsto_{\eta^\infty} w^\infty$. Hence, the conclusion follows.
- If $\lambda x.e \leadsto_{\eta_i} f_i$ then, for all v $e \leadsto_{\eta_i[x\to v]} f_i(v)$. By inductive hypothesis, $e \leadsto_{\eta^\infty[x\to v]} f^\infty(v)$. Hence, $\lambda x.e \leadsto_{\eta^\infty} f^\infty$.

■

Theorem 3.7 (Completeness) For all expressions e^τ and all environments η there is a $v \in \mathcal{D}_\tau$ such that $e \leadsto_{\tau,\eta} v$ and $abs_\tau(v) = [\![e]\!]^{\mathbf{I}}\eta^{\mathbf{I}}$.

We use the method of logical relations [1, 7, 12, 15] to prove this. Define a family of predicates P_τ as follows:

- $P_\beta(e^\beta)$ iff, for all environments η, there is a $v \in D_\beta$ such that $e \leadsto_{\beta,\eta} v$ and $abs_\beta(v) = [\![e]\!]^{\mathbf{I}}\eta^{\mathbf{I}}$.
- $P_{\tau_1\times\tau_2}(e)$ iff $P_{\tau_1}(e.1)$ and $P_{\tau_2}(e.2)$.
- $P_{\tau_1\to\tau_2}(e)$ iff, for all e', $P_{\tau_1}(e') \Rightarrow P_{\tau_2}(ee')$.

We first show that $P(e)$ implies that e satisfies the required property. (This automatically holds for base types by definition, but not for higher types). Next, we show that all expressions e satisfy P. The logical relation is essentially the strengthened inductive hypothesis needed for this latter proof. It then follows that all expressions satisfy the required property.[2]

Lemma 3.8 If $P_\tau(e^\tau)$ then, for all environment η, there is a $v \in \mathcal{D}_\tau$ such that $e \rightsquigarrow_{\tau,\eta} v$ and $abs_\tau(v) = [\![e]\!]^{\mathbf{I}}\eta^{\mathbf{I}}$.

Proof: By structural induction on the type τ:

- Case $\tau = \beta$ follows from definition.
- Case $\tau = \tau_1 \times \tau_2$. Assume $P_{\tau_1 \times \tau_2}(e)$, which means $P_{\tau_1}(e.1)$ and $P_{\tau_2}(e.2)$. By inductive hypothesis, there are values v_1 and v_2 such that $e.i \rightsquigarrow_{\tau_i,\eta} v_i$ and $abs(v_i) = [\![e.i]\!]^{\mathbf{I}}\eta^{\mathbf{I}}$. Let $v = \langle v_1, v_2\rangle$. By definition of $\rightsquigarrow$, $e \rightsquigarrow_{\tau_1\times\tau_2,\eta} v$. Further, $abs(\langle v_1, v_2\rangle = [\![e]\!]^{\mathbf{I}}\eta^{\mathbf{I}}$.
- Case $\tau = \tau_1 \rightarrow \tau_2$. Assume $P_{\tau_1\rightarrow\tau_2}(e)$, which means $\forall e_1.\ P_{\tau_1}(e_1) \Rightarrow P_{\tau_2}(ee_1)$. Let $e_1 = x$ be a variable that is not free in e, and assume $P_{\tau_1}(x)$. By inductive hypothesis for τ_2, we have

  $$\forall\eta.\ \forall v \in D_{\tau_1}.\ \exists w \in D_{\tau_2}.\ ex \rightsquigarrow_{\tau_2,\eta[x\rightarrow v]} w \wedge abs(w) = [\![e]\!]^{\mathbf{I}}\eta^{\mathbf{I}}(abs(v))$$

  For every η, this defines a function f mapping v's to w's. By definition of $\rightsquigarrow$, $e \rightsquigarrow_{\tau_1\rightarrow\tau_2,\eta} f$. Secondly, $abs_{\tau_1\rightarrow\tau_2}(f) = \lambda a.\ \sqcup\{abs(f(v)) \mid abs(v) \sqsubseteq a\}$. Further, for abstract values in the codomain of abs,

  $$\begin{aligned} abs(f)(abs(v)) &= \sqcup\{abs(f(v')) \mid abs(v') = abs(v)\} \\ &= \sqcup\{[\![e]\!]^{\mathbf{I}}\eta^{\mathbf{I}}(abs(v')) \mid abs(v') = abs(v)\} \\ &= [\![e]\!]^{\mathbf{I}}\eta^{\mathbf{I}}(abs(v)) \end{aligned}$$

  Hence, $abs(f) = [\![e]\!]^{\mathbf{I}}\eta^{\mathbf{I}}$.

■

Lemma 3.9 Let a *neutral term* be one of x, k, $t.1$, $t.2$ and te where t is a neutral term and e any term such that $P(e)$. Then, $P_\tau(t^\tau)$ for all neutral terms t^τ.

Proof: By structural induction on τ:

[2] In the following proof, we ignore the case of *fix* terms which involve a lengthier analysis.

- Case $\tau = \beta$. For every neutral term t^β, there exists a v such that $t \sim_{\beta,\eta} v$ and $abs(v) = [\![t]\!]^\natural \eta^\natural$. This may be shown by structural induction on t.
- Case $\tau = \tau_1 \times \tau_2$. By inductive hypothesis, we have $P_{\tau_1}(t.1)$ and $P_{\tau_2}(t.2)$. Hence, the conclusion.
- Case $\tau = \tau_1 \to \tau_2$. Let e be a term such that $P_{\tau_1}(e)$. By inductive hypothesis, $P_{\tau_2}(te)$. Hence, the conclusion follows.

■

Lemma 3.10 If e is an expression with free variables $x_1, \ldots, x_n$ and $t_1, \ldots, t_n$ expressions satisfying P then $P(e[t_1/x_1, \ldots, t_n/x_n])$.

Proof: By structural induction on e:

- If $e = x_i$, then t_i satisfies P by assumption.
- If $e = k$, we have the conclusion by Lemma 3.9.
- If $e = \langle e_1, e_2 \rangle$, by inductive hypothesis $P(e_1[\bar{t}/\bar{x}])$ and $P(e_2[\bar{t}/\bar{x}])$. Hence, by definition of P for product types, $P(e_1[\bar{t}/\bar{x}]e_2[\bar{t}/\bar{x}])$.
- If $e = e'.1$, by inductive hypothesis, $P(e'[\bar{t}/\bar{x}])$. By definition of P for product types, this implies $P(e'.1[\bar{t}/\bar{x}])$.
- If $e = \lambda y.e'$, by inductive hypothesis $P(e'[\bar{t}/\bar{x}, u/y])$ for every expression u such that $P(u)$. By definition of P for function types, this means $P(\lambda y.e'[\bar{t}/\bar{x}])$.
- If $e = e_1 e_2$, by inductive hypothesis, $P(e_1[\bar{t}/\bar{x}])$ and $P(e_2[\bar{t}/\bar{x}])$. $P(e_1[\bar{t}/\bar{x}])$ means, for all u, $P(u) \Rightarrow P(e_1[\bar{t}/\bar{x}]u)$. Hence $P(e[\bar{t}/\bar{x}])$.

■

Theorem 3.7 now follows from the three lemmas. Since a variable x_i is a neutral term we have $P(x_i)$ by Lemma 3.9. Then, use Lemma 3.10 to observe $P(e[\bar{x}/\bar{x}])$ for all e. Finally, Lemma 3.8 ensures the conclusion. □

4 Conclusion

We provide here a general framework for discussing the completeness issues of static analysis using abstract interpretation. The essential idea is that abstract interpretation blurs certain distinctions in

the concrete values by mapping them to abstract values. If the concrete values mapped to identical abstract values are considered indistinguishable, we can raise the question whether the analysis can detect all the properties shared by indistinguishable classes of programs.

Our results give an affirmative answer for two cases: For the analysis of first-order function definitions using a totally ordered abstract domain, the abstract interpretation is complete. For higher-order functional programs with pairs, the abstract interpretation is complete if structured values considered indistinguishable whenever their components are indistinguishable.

We also show that the results do not generalize to arbitrarily ordered abstract domains. Essentially, the results fail because the least upper bounds of abstract values can involve loss of information. It should be possible to formalize such loss of information in a precise way and duplicate our results for that case. This may be able to cover, for example, the abstraction of sum domains by products of abstract domains.

Forward abstract interpretation cannot give certain essential information about program evaluation such as the property of head-strictness [17]. The analysis for such properties can be characterized as the abstract interpretation of the continuation semantics of programs. It would be interesting to check if such backward analysis has completeness properties of the kind investigated here.

References

[1] ABRAMSKY, S. Abstract interpretation, logical relations, and kan extensions. *J. of Logic and Computation 1*, 1 (Jul 1990), 5–40.

[2] ABRAMSKY, S., AND HANKIN, C. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[3] BURN, G., HANKIN, C., AND ABRAMSKY, S. The theory of strictness analysis for higher order functions. In *LNCS 217*, Springer-Verlag, Berlin, 1985, pp. 42–62.

[4] DAMAS, L., AND MILNER, R. Principal type-schemes for functional programs. In *ACM Symp. on Princ. of Program. Languages* (1982), pp. 207–212.

[5] DEBRAY, S. K. Static inference of modes and data dependencies in logic programs. *ACM Trans. Program. Lang. Syst. 11*, 3 (July 1989), 418–450.

[6] HUNT, S. *PERs Generalise Projections for Strictness Analysis.* Research Report DOC 90/14, Imperial College, London, Oct 1990.

[7] MITCHELL, J. C. Type systems for programming languages. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed., North-Holland, Amsterdam, 1990, ch. , p. . (also Report No. STAN-CS-89-1277, Department of Computer Science, Stanford University).

[8] MITCHELL, J. C., AND HARPER, R. The essence of ML. In *ACM Symp. on Princ. of Program. Languages* (1988), ACM, pp. 28–46.

[9] MYCROFT, A. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* PhD thesis, Univ. of Edinburgh, Dec. 1981.

[10] MYCROFT, A. The theory and practice of transforming call-by-need into call-by-value. In *4th Intl. symp. on Programming*, Springer-Verlag, 1980, pp. 269–281.

[11] NIELSON, F. Expected forms of data flow analyses. In *Programs as Data Objects*, H. Ganzinger and N. D. Jones, Eds., Springer-Verlag, Berlin, 1985, pp. 172–191. (LNCS Vol. 217).

[12] PLOTKIN, G. D. Lambda definability in the full type hierarchy. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, J. R. Hindley and J. P. Seldin, Eds., Academic Press, New York, 1980, pp. 363–373.

[13] SEKAR, R. C., MISHRA, P., AND RAMAKRISHNAN, I. V. On the power and limitation of strictness analysis based on abstract interpretation. In *ACM Symp. on Princ. of Program. Languages* (Orlando, Fl., Jan 1991), ACM, pp. 37–48.

[14] SEKAR, R. C., PAWAGI, S., AND RAMAKRISHNAN, I. V. Small domains spell fast strictness analysis. In *ACM Symp. on Princ. of Program. Languages* (San Francisco, CA., Jan 1990), ACM, pp. 169–183.

[15] STATMAN, R. Logical relations and the typed lambda calculus. *Information and Control 65* (1985), 85–97.

[16] WADLER, P. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In *Abstract Interpretation of Declarative Languages*, S. Abramsky and C. Hankin, Eds., Ellis Horwood, 1987, p. .

[17] WADLER, P., AND HUGHES, R. J. M. Projections for strictness analysis. In *Conf. on Functional Program. Lang. and Comput. Arch.*, G. Kahn, Ed., Springer-Verlag, Berlin, 1987, pp. 385–407.

S22-61
94021
8P,
0745/44

Head-strictness is not a monotonic abstract property

Samuel Kamin

Computer Science Dept.
University of Illinois at Urbana-Champaign
Urbana, IL 61801

August 12, 1991

Abstract

A property $\mathcal{P}$ of a language is said to be ***definable by abstract interpretation*** if there is a monotonic map **abs** from the domain of standard semantics to an abstract domain $\mathcal{A}$ of finite height, and a partition of the abstract domain into two parts $\mathcal{A}_{\mathcal{P}}$ and $\mathcal{A}_{\text{non-}\mathcal{P}}$, such that any value has property $\mathcal{P}$ if and only if **abs** maps it to an element of $\mathcal{A}_{\mathcal{P}}$. ***Head-strictness*** is a property of functions over lists which asserts, roughly speaking, that whenever the function looks at a the tail of a list, it looks at the head of the tail. We prove that head-strictness is not definable by abstract interpretation. We then present a non-monotonic abstract interpretation for head-strictness and prove its safety.

Keywords: **strictness analysis, abstract interpretation, lazy evaluation**

To appear: Information Processing Letters

1 Introduction

Given a programming language $\mathcal{L}$ with standard semantics $[\![\cdot]\!]_{\mathcal{D}} : \mathcal{L} \to \mathcal{D}$, an *abstract interpretation* [2, 6] of $\mathcal{L}$ is a semantics $[\![\cdot]\!]_{\mathcal{A}} : \mathcal{L} \to \mathcal{A}$, together with a function **abs** : $\mathcal{D} \to \mathcal{A}$. $\mathcal{A}$ represents some property of programs in $\mathcal{L}$, $[\![\cdot]\!]_A$ is the static computation of that property, and **abs** is the definition of that property with respect to semantic values. In general, $[\![\cdot]\!]_A$ can only be an approximation to **abs** $\circ$ $[\![\cdot]\!]_D$, due to computability considerations. Despite this, the property may be considered to be subject to analysis by abstract interpretation. Strictness of functions (the property that $f\perp = \perp$) is an example of such a property [4, 5, 7].

Under what circumstances, then, can we assert that a property is *not* analyzable by abstract interpretation? If there are no constraints on $\mathcal{A}$ or **abs**, we can never say it. However, two constraints are commonly placed on them:

1. $\mathcal{A}$ is a domain of finite height. This ensures that $[\![\cdot]\!]_A$ can be computed even if it involves fixedpoints.

2. **abs** is monotonic. This permits properties of fixedpoints in $\mathcal{D}$ to be inferred from the corresponding fixedpoints in $\mathcal{A}$.

Thus, we will say:

Definition 1 *A property $\mathcal{P}$ of $\mathcal{L}$ is* definable by abstract interpretation *if there is an abstract interpretation as above, where $\mathcal{A}$ is of finite height and* **abs** *is monotonic, and a partition of $\mathcal{A}$ into $\mathcal{A}_{\mathcal{P}} \cup \mathcal{A}_{non\text{-}\mathcal{P}}$, such that for any program ℓ, ℓ has property $\mathcal{P}$ if and only if* $\mathbf{abs}([\![\ell]\!]_{\mathcal{D}}) \in \mathcal{A}_{\mathcal{P}}$.

We propose this as the most general possible reading for "definable by abstract interpretation." Note that it makes no substantive reference to $[\![\cdot]\!]_{\mathcal{A}}$: the ability to actually compute $\mathcal{P}$ is not at issue. We are simply saying that, at the minimum, we must be able to define $\mathcal{P}$ by a monotonic map and "read off" the answer from $\mathcal{A}$.

Given this definition, non-definability of property $\mathcal{P}$ can be proven by producing an infinite sequence of elements of $\mathcal{D}$ that alternate between $\mathcal{P}$ and non-$\mathcal{P}$.

Head-strictness [3, 11] is a property of functions of lists that has for some time been thought not to be definable by abstract interpretation. What it says, intuitively, is that if the function examines any tail of its argument, it evaluates all the heads of the list up to the head of that tail. Specifically:

Definition 2 *Function f is* head-strict, *or* calH-strict, *if f is strict and $f \circ \mathcal{H} = f$, where $\mathcal{H}$ is defined by:*

$$\begin{array}{rlll} \mathcal{H}: & \perp & \mapsto & \perp \\ & nil & \mapsto & nil \\ & \perp.t & \mapsto & \perp \\ & h.t & \mapsto & h.\mathcal{H}(t), \ \text{if } h \neq \perp \end{array}$$

Our main theorem is:

Theorem 1 *Head-strictness is not definable by abstract interpretation.*

As an easy corollary of the proof of theorem 1, we can obtain a proof of undefinability of a useful variant of head-strictness:

Definition 3 *f is* head-strict in a head-strict context *if f is strict and $\mathcal{H} \circ f \circ \mathcal{H} = \mathcal{H} \circ f$.*

Theorem 2 ***Head-strictness in a head-strict context is not definable by abstract interpretation.***

Section 2 contains the proofs of theorems 1 and 2, and also discusses "undefinability by BHA-style analysis" [1, 4]. Section 3 presents a non-monotonic abstract interpretation for head-strictness.

2 Head-strictness is not definable

$\mathcal{H}$ is an example of a *projection*, that is, an idempotent function below the identity in the function space ordering. In [11], strictness properties of functions on lists are formulated using projections; head-strictness and head-strictness in a head-strict context are examples.

Given a projection $\alpha : \mathcal{D} \to \mathcal{D}$, define a value x to be *α-strict* if $\alpha(x) = x$. Many projections used in strictness analysis are *smash projections* [4], which means they map every element either to $\bot$ or to itself. Such projections lead to properties having a simple structure:

Definition 4 *Given projections α and β and function f, we say f* has property *$\alpha \Rightarrow \beta$, or write $f : \alpha \Rightarrow \beta$, if f is strict and $\alpha \circ f = \alpha \circ f \circ \beta$.*

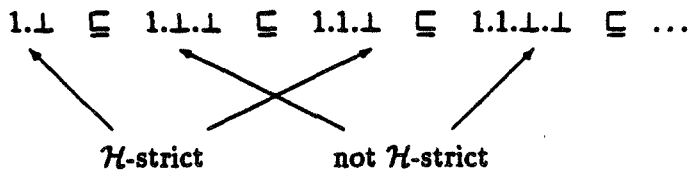
Thus, head-strictness is property $\mathcal{I} \Rightarrow \mathcal{H}$, where $\mathcal{I}$ is the identity function, and head-strictness in a head-strict context is property $\mathcal{H} \Rightarrow \mathcal{H}$.

Fact 1 ***If α and β are smash projections, then property $\alpha \Rightarrow \beta$ is downward closed.***

Proof Our assumptions are that f is strict, $\alpha \circ f = \alpha \circ f \circ \beta$, and $g \sqsubseteq f$. Obviously, g is strict. Then, for any x, there are two possibilities:

- $\beta x = \bot$. Then $g \sqsubseteq f$ implies $\alpha(gx) \sqsubseteq \alpha(fx) = \alpha(f(\beta x)) = \alpha(f\bot) = \bot$, so $\alpha(gx) = \bot$. But $\alpha(g(\beta x)) \sqsubseteq \alpha(gx)$, so $\alpha(g(\beta x)) = \bot$ also.
- $\beta x = x$. In this case, $\alpha(g(\beta x)) = \alpha(gx)$ is immediate.

The simple structure of the property $\alpha \Rightarrow \beta$ in this case allows it to be captured by a monotonic function. The peculiar feature of $\mathcal{H}$ is that it is not a smash projection. In fact, we can construct an infinite sequence of lists that are alternately $\mathcal{H}$-strict and non-$\mathcal{H}$-strict:



where 1 is some value of $\mathcal{D}$ other than $\perp$. This observation can be parlayed into a proof of theorem 1:

Proof of theorem 1 As stated earlier, it suffices to present an increasing sequence of functions $< f_i >_{i \geq 0}$, such that for all i, f_i is $\mathcal{H}$-strict *iff* f_{i+1} is not $\mathcal{H}$-strict. The result follows from the observation that these functions are computable and therefore contained in $\mathcal{D}$.

Notation $x^i.y$ is the list $\underbrace{x.x.\ldots.x}_{i \text{ times}}.y$

Continuing to assume 1 is an element of $\mathcal{D}$ other than $\perp$, define the function g from lists to lists:

$$\begin{array}{rcl} g & : & \perp \mapsto \perp \\ & : & x \mapsto 1.\perp, \ x \neq \perp \end{array}$$

Now define the sequence $< f_i >_i$ by, for all $i \geq 0$:

$$\begin{array}{rcl} f_{2i} & : & \perp \mapsto \perp \\ & : & \perp.x \mapsto \perp \\ & : & x.y \mapsto 1^i.\perp, \ x \neq \perp \\ f_{2i+1} & : & \perp \mapsto \perp \\ & : & \perp.x \mapsto \perp \\ & : & x.y \mapsto 1^i.g(y), \ x \neq \perp \end{array}$$

In other words, $f_{2i}(x.y)$ inspects x and, if the inspection ever terminates, returns the list $1^i.\perp$. $f_{2i+1}(x.y)$ inspects x and return either $1^i.\perp$ or $1^{i+1}.\perp$, depending upon the result of inspecting y; in programming terms, it returns $1^i.t$, where t is a thunk which, when evaluated, inspects y and, if that inspection terminates, returns $1.\perp$. Clearly, all these functions are computable.

That $< f_i >_i$ forms a non-decreasing sequence is obvious. We need only:

1. f_{2i} is $\mathcal{H}$-strict. Consider the cases:

$$\begin{array}{lcl} f_{2i}(\perp) & = & f_{2i}(\mathcal{H}(\perp)) \\ f_{2i}(\perp.x) & = & \perp \ = \ f_{2i}(\perp) \ = \ f_{2i}(\mathcal{H}(\perp.x)) \\ f_{2i}(x.y) & = & 1^i.\perp \ = \ f_{2i}(x.\mathcal{H}(y)) \quad \text{(since } f_{2i}(x.y) \text{ does not depend on } y\text{)} \\ & = & f_{2i}(\mathcal{H}(x.y)) \end{array}$$

2. f_{2i+1} is not $\mathcal{H}$-strict. Consider the list $1.\perp.\perp$:

$$\begin{array}{lcl} f_{2i+1}(1.\perp.\perp) & = & 1^{i+1}.\perp \\ f_{2i+1}(\mathcal{H}(1.\perp.\perp)) & = & f_{2i+1}(1.\perp) \ = \ 1^i.\perp \end{array}$$

Thus, $f_{2i+1} \neq f_{2i+1} \circ \mathcal{H}$.

As an immediate corollary, we obtain:

Proof of theorem 2 Recall that $f : \mathcal{H} \Rightarrow \mathcal{H}$ if f is strict and $\mathcal{H} \circ f \circ \mathcal{H} = \mathcal{H} \circ f$. Then simply note that $\mathcal{H} \circ f_i = f_i$ for all i in the proof of theorem 1. It follows that for all i, f_{2i} is head-strict in a head-strict context and f_{2i+1} is not.

2.1 BHA-style analysis

One formulation of strictness analysis that is widely followed is the one presented in [1, 4]. There, the Scott-closed subsets of $\mathcal{D}$ — the relevant property of these subsets is that they are downward-closed — play a leading role. In particular, BHA-style analyses can only characterize properties that are downward-closed, such as strictness. Thus, the existence of the chain $f_1 \sqsubseteq f_2$ suffices to prove the undefinability of head-strictness, since f_2 is head-strict and f_1 is not; adding f_0 demonstrates the undefinability of *non*-head-strictness.

It is probably fair to say that most abstract interpretations of interest are either downward-closed or upward-closed. That is, most program transformations are either equivalence-preserving in the presence of non-termination or equivalence-preserving in the presence of guaranteed termination. ([1] extends the BHA-style analysis to upward-closed properties.) However, this still leaves many properties occupying an undefinable middle ground — most obviously, any product of a downward-closed and an upward-closed analysis. Our result is stronger than undefinability via BHA-style analysis, since it precludes analysis by any method whatsoever.

3 Non-monotonic abstract interpretations

Recently, Hunt [8], building on [1], has presented a non-monotonic abstract interpretation for head-strictness, basing his safety proof on logical relations. In this section, we present our own and prove its safety from elementary principles.

For our abstract domain $\mathcal{A}$, we choose the powerset of strictness properties $\alpha \Rightarrow \beta$, after first extending them to multiple arguments.

Definition 5 *Given functions $\beta_1 : A_1 \to B_1, \ldots, \beta_n : A_n \to B_n$, $\vec{\beta} : A_1 \times \cdots \times A_n \to B_1 \times \cdots \times B_n$ is the function defined by: $\vec{\beta}(a_1, \ldots, a_n) = (\beta_1(a_1), \ldots, \beta_n(a_n))$.*

Definition 6 *Given a vector $\vec{\beta}$ of n projections and a projection α, we write $f : \alpha \Rightarrow \vec{\beta}$, for n-argument function f, when $\alpha \circ f = \alpha \circ f \circ \vec{\beta}$ and f is strict (i.e. $f(\bot, \ldots, \bot) = \bot$).*

We will assume we have chosen a finite set of projections which includes $\mathcal{I}$, and will denote by ST the set of all strictness properties $\alpha \Rightarrow \vec{\beta}$ over them. Then define

$$\mathcal{A} = \mathcal{P}(ST),$$

the power set of ST, ordered by containment.

Our (non-monotonic) abstraction function is the obvious one:

$$\mathbf{abs}(f) = \{\alpha \Rightarrow \vec{\beta} \mid f : \alpha \Rightarrow \vec{\beta}\}$$

Note that $\mathbf{abs}(f)$ is necessarily non-empty, since it contains $\alpha \Rightarrow \mathcal{I}$ for all α.

Definition 7 *Define the abstract semantics $[\![\cdot]\!]_{\mathcal{A}}$ taking expressions to functions from a function environment η to an element of $\mathcal{A}$. We assume that the expression contains only the variables $x_1, \ldots, x_n$ (this is notationally simpler than using a value environment). η tells us the strictness of functions appearing in the expression:*

$$
\begin{array}{lcl}
[\![c]\!]_{\mathcal{A}}\eta & = & ST \\
[\![x_i]\!]_{\mathcal{A}}\eta & = & \{\alpha \Rightarrow \vec{\beta} \mid \beta_i = \alpha\} \\
[\![(\mathit{if}\, e_1\, e_2\, e_3)]\!]_{\mathcal{A}}\eta & = & \{\alpha \Rightarrow \vec{\beta} \mid \mathcal{I} \Rightarrow \vec{\beta} \in [\![e_1]\!]_{\mathcal{A}}\eta\} \cap [\![e_2]\!]_{\mathcal{A}}\eta \cap [\![e_3]\!]_{\mathcal{A}}\eta \\
[\![(f\, e_1 \ldots e_3)]\!]_{\mathcal{A}}\eta & = & \bigcup\{\alpha \Rightarrow \vec{\beta} \mid \exists \vec{\gamma}.\ \alpha \Rightarrow \vec{\gamma} \in \eta(f)\ \&\ \gamma_i \Rightarrow \beta \in [\![e_i]\!]_{\mathcal{A}}\eta \textit{ for all } i\}
\end{array}
$$

Theorem 3 *Our analysis is safe in the sense that:*

$$[\![e]\!]_{\mathcal{A}}\eta \subseteq \mathbf{abs}(\lambda\vec{x}.[\![e]\!]_{\mathcal{D}}\eta_{\mathcal{D}}),$$

assuming η is safe in the sense that for all f, $\eta(f) \subseteq \mathbf{abs}(\eta_{\mathcal{D}}(f))$.

Proof By induction. Here, every expression is taken implicitly to be a function of variables $x_1, \ldots, x_n$. To avoid clutter, we will write $[\![e]\!]$ instead of $\lambda\vec{x}.[\![e]\!]_{\mathcal{D}}\eta_{\mathcal{D}}$.

Base cases are variables and constants:

- $[\![c]\!] = \lambda\vec{x}.c$:

$$
\begin{array}{lcl}
(\alpha \circ [\![c]\!] \circ \vec{\beta})(\vec{x}) & = & \alpha([\![c]\!](\vec{\beta}(\vec{x}))) \\
& = & \alpha(c) \\
& = & \alpha([\![c]\!](\vec{x})) \\
& = & (\alpha \circ [\![c]\!])(\vec{x})
\end{array}
$$

  Since we made no assumptions about α or $\vec{\beta}$, every $\alpha \Rightarrow \vec{\beta} \in \mathbf{abs}([\![c]\!])$.

- $[\![x_i]\!] = \lambda\vec{x}.x_i$ and $\alpha \Rightarrow \vec{\beta} \in [\![x_i]\!]_{\mathcal{A}}$ iff $\beta_i = \alpha$:

$$
\begin{array}{lcl}
(\alpha \circ [\![x_i]\!] \circ \vec{\beta})(\vec{x}) & = & \alpha([\![x_i]\!](\vec{\beta}(\vec{x}))) \\
& = & \alpha(\vec{\beta}(\vec{x})_i) \\
& = & \alpha(\beta_i(x_i)) \\
& = & \alpha(\alpha(x_i)) \\
& = & \alpha(x_i) \\
& = & (\alpha \circ [\![x_i]\!])(\vec{x})
\end{array}
$$

The induction cases are *if* and function application:

- $[\![(\mathit{if}\ e_1\ e_2\ e_3)]\!] = \lambda\vec{x}.$ if $[\![e_1]\!]\vec{x}$ then $[\![e_2]\!]\vec{x}$ else $[\![e_3]\!]\vec{x}$, so,

$$(\alpha \circ [\![(\mathit{if}\ e_1\ e_2\ e_3)]\!] \circ \vec{\beta})(\vec{x}) = \text{if } [\![e_1]\!](\vec{\beta}(\vec{x})) \text{ then } \alpha([\![e_2]\!](\vec{\beta}(\vec{x}))) \text{ else } \alpha([\![e_3]\!](\vec{\beta}(\vec{x})))$$
$$(\alpha \circ [\![(\mathit{if}\ e_1\ e_2\ e_3)]\!])(\vec{x}) = \text{if } [\![e_1]\!]\vec{x} \text{ then } \alpha([\![e2]\!]\vec{x}) \text{ else } \alpha([\![e3]\!]\vec{x})$$

  These are equal if both of the following hold:

  1. $[\![e_1]\!](\vec{\beta}(\vec{x})) = [\![e_1]\!]\vec{x}$, which is the case if $\mathcal{I} \Rightarrow \vec{\beta} \in [\![e_1]\!]_{\mathcal{A}}\eta$.
  2. $\alpha([\![e_i]\!](\vec{\beta}(\vec{x}))) = \alpha([\![e_i]\!]\vec{x})$, for $i = 2, 3$, which is true if $\alpha \Rightarrow \vec{\beta} \in [\![e_i]\!]_{\mathcal{A}}\eta$.

Thus, $[\![(if\ e_1\ e_2\ e_3)]\!]_{\mathcal{A}}\eta$ contains $\alpha \Rightarrow \vec{\beta}$, so long as both e_2 and e_3 contain it, and e_1 can return a truth value even after applying $\vec{\beta}$ to its arguments.

- $[\![(f\ e_1 \cdots e_m)]\!]\vec{x} = \eta_{\mathcal{D}}(f)([\![e_1]\!]\vec{x}, \cdots, [\![e_m]\!]\vec{x})$. By assumption, $\eta_{\mathcal{D}}(f)$ satisfies the hypothesis. Now consider:

$$\alpha([\![(f\ e_1 \cdots e_n)]\!](\vec{\beta}(\vec{x}))) = \alpha(\eta_{\mathcal{D}}(f)([\![e_1]\!](\vec{\beta}(\vec{x})), \cdots, [\![e_m]\!](\vec{\beta}(\vec{x}))))$$

Suppose $\exists\vec{\gamma}$ such that $\alpha \Rightarrow \vec{\gamma} \in \eta_{\mathcal{D}}(f)$ and $\forall i, \gamma_i \Rightarrow \vec{\beta} \in [\![e_i]\!]_{\mathcal{A}}$. Then,

$$\begin{aligned}
\alpha(\eta_{\mathcal{D}}(f)([\![e_1]\!](\vec{\beta}(\vec{x})), \ldots)) &= \alpha(\eta_{\mathcal{D}}(f)(\vec{\gamma}([\![e_1]\!](\vec{\beta}(\vec{x})), \ldots))) \\
&= \alpha(\eta_{\mathcal{D}}(f)(\gamma_1([\![e_1]\!](\vec{\beta}(\vec{x}))), \ldots)) \\
&= \alpha(\eta_{\mathcal{D}}(f)(\gamma_1([\![e_1]\!]\vec{x}), \ldots)) \\
&= \alpha(\eta_{\mathcal{D}}(f)(\vec{\gamma}([\![e_1]\!]\vec{x}, \ldots))) \\
&= \alpha(\eta_{\mathcal{D}}(f)([\![e_1]\!]\vec{x}, \ldots))
\end{aligned}$$

The key remaining issue is the safety of fixedpoints. This is crucial because it is precisely why monotonicity is ordinarily required.

Suppose, then, that we have a recursive definition

$$f\vec{x} = e,$$

where e has no variables other than $\vec{x}$. For notational simplicity, we will assume that e calls no functions other than f, so this definition simply denotes a function (rather than an environment). The standard and abstract semantics of the definition are:

$$[\![f\vec{x} = e]\!]_{\mathcal{D}} = \mathbf{fix}(\lambda\phi_{\mathcal{D}}.[\![e]\!]_{\mathcal{D}}\{f \to \phi_{\mathcal{D}}\})$$

$$[\![f\vec{x} = e]\!]_{\mathcal{A}} = \mathbf{fix}(\lambda\phi_{\mathcal{A}}.[\![e]\!]_{\mathcal{A}}\{f \to \phi_{\mathcal{A}}\})$$

We need that $[\![f\vec{x} = e]\!]_{\mathcal{A}} \subseteq \mathbf{abs}([\![f\vec{x} = e]\!]_{\mathcal{D}})$. The bottom element in $\mathcal{A}$ is $\emptyset$, while in $\mathcal{D}$ we will denote it as $\perp$. Obviously, $\{f \to \emptyset\}$ is safe with respect to $\{f \to \perp\}$. Furthermore, theorem 3 allows us to conclude that for any i, $[\![e]\!]^i_{\mathcal{A}}\{f \to \emptyset\} \subseteq \mathbf{abs}([\![e]\!]^i_{\mathcal{D}}\{f \to \perp\})$. But $\bigsqcup_i\{[\![e]\!]^i_{\mathcal{A}}\{f \to \emptyset\}\} = [\![e]\!]^j_{\mathcal{A}}\{f \to \emptyset\}$ for some j, since $\mathcal{A}$ is of finite height. So we only need the following:

Lemma 1 *If $S \subseteq \mathbf{abs}([\![e]\!]^i_{\mathcal{D}}\eta)$ for all i, then $S \subseteq \mathbf{abs}(\bigsqcup_i\{[\![e]\!]^i_{\mathcal{D}}\eta)\}$.*

Proof Suppose $\alpha \Rightarrow \vec{\beta} \in S$. Then

$$\begin{aligned}
\alpha \circ \bigsqcup_i([\![e]\!]^i\eta) \circ \vec{\beta} &= \bigsqcup_i \alpha \circ ([\![e]\!]^i\eta) \circ \vec{\beta}, \text{ by continuity of composition} \\
&= \bigsqcup_i \alpha \circ ([\![e]\!]^i\eta), \text{ since } \alpha \Rightarrow \vec{\beta} \in \mathbf{abs}([\![e]\!]^i\eta) \\
&= \alpha \circ \bigsqcup_i([\![e]\!]^i\eta), \text{ by continuity of composition.}
\end{aligned}$$

Acknowledgements

A number of people suggested improvements to an earlier version of this paper. Aside from the referees, who were most helpful, I'd like to thank Geoffrey Burn, Marc Neuberger, and Prateek Mishra. Thanks also to Uday Reddy for many interesting conversations about strictness.

References

[1] S. Abramsky, "Abstract interpretation, logical relations, and Kan extensions," J. Logic and Computation 1(1), July 1990, 5–40.

[2] S. Abramsky, C.L. Hankin (eds.), *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987,

[3] G. Burn, "A relationship between abstract interpretation and projection analysis (extended abstract)," 17th POPL, 1990, 151–156.

[4] G. Burn, C.L. Hankin, S. Abramsky, "Strictness analysis of higher-order functions," *Science of Computer Programming* 7, Nov. 1986, 249–278.

[5] C. Clack, S.L. Peyton Jones, "Strictness analysis — a practical approach," Conf. on Functional Programming Languages and Computer Architecture, 1985, LNCS 201, 35–49.

[6] P. Cousot, R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixed points," 4th POPL, 1977, 238–252.

[7] P. Hudak, J. Young, "Higher-order strictness analysis in untyped lambda calculus," 13th POPL, 1986, 97–109.

[8] S. Hunt, "PERs generalize projections for strictness analysis," Glasgow Workshop on Functional Programming, Ullapool, in *Workshops in Computing*, Springer-Verlag, 1991.

[9] S. L. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice-Hall International, New York, 1987.

[10] P. Wadler, "Strictness analysis on non-flat domains," Chapter 12 in [2], 266–275.

[11] P. Wadler, R.J.M. Hughes, "Projections for strictness analysis," Conf. on Functional Programming Languages and Computer Arch., 1987, LNCS 274, 385–407.

Assignments for Applicative Languages

Vipin Swarup[*] — The MITRE Corporation

Uday S. Reddy[†] — University of Illinois at Urbana-Champaign

Evan Ireland[‡] — Massey University

Abstract

We propose a theoretical framework for adding assignments and dynamic data to functional languages without violating their semantic properties. This differs from semi-functional languages like Scheme and ML in that values of expressions remain static and side-effect-free. A new form of abstraction called *observer* is designed to encapsulate state-oriented computation from the remaining purely applicative computation. The type system ensures that observers are combined linearly, allowing an implementation in terms of a global store. The utility of this extension is in manipulating *shared dynamic data* embedded in data structures. Evaluation of well-typed programs is Church-Rosser. Thus, programs produce the same results whether an eager or lazy evaluation order is used (assuming termination). A simple, sound logic permits reasoning about well-typed programs. The benefits of this work include greater expressive power and efficiency (compared to applicative languages), while retaining simplicity of reasoning.

Keywords Functional languages, imperative programming, lambda calculus, type systems, strong normalization, Church-Rosser property, referential transparency, continuation-passing style.

1 Introduction

Functional languages are popular among computer scientists because of their strong support of modularity. They possess two powerful glues, higher-order functions and laziness, that permit programs to be modularized in new, useful ways. Hughes [Hug90] convincingly argues that "...lazy evaluation is too important to be relegated to second-class citizenship. It is perhaps the most powerful glue functional programmers possess. One should not obstruct access to such a vital tool." However, side-effects are incompatible with laziness: programming with them requires knowledge of global context, defeating the very modularity that lazy evaluation is designed to enhance.

[*] Address: The MITRE Corporation, Burlington Road, Bedford, MA 01730; E-mail: swarup@mitre.org;
Supported by NASA grant NAG-1-613 (while at the University of Illinois at Urbana-Champaign).
[†] Address: Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801;
E-mail: reddy@cs.uiuc.edu; Supported by a grant from Motorola Corporation.
[‡] Address: School of Information Sciences, Massey University, Palmerston North, New Zealand;
E-mail: E.Ireland@massey.ac.nz

Functional Programming Languages and
Computer Architecture, Springer-Verlag LNCS 523

Pure functional languages have nice properties that make them easy to reason about. For instance, + is commutative, = is reflexive, and most other familiar mathematical properties hold of the computational operators. This is a consequence of expressions representing *static* values: values that do not change over time. Thus, an expression's value is independent of the order in which its subexpressions are evaluated. Side-effects are incompatible with these properties, as side-effects change the values of other expressions, making the order of evaluation important.

Assignments are a means of describing *dynamic* data : data whose values change over time. In their conventional form, assignments have side-effects on their environment, making their order of evaluation important. Not only are such assignments incompatible with laziness, but they also destroy the nice mathematical properties of pure languages. Hence lazy functional languages shun assignments.

However, since assignments directly model the dynamic behavior of a physical computer's store, they yield efficient implementations of dynamic data. In contrast, one models dynamic data in functional languages by representing the state explicitly or, possibly, by creating streams of states. Compilation techniques and language notations have been proposed to permit explicit state manipulation to be implemented efficiently [HB85, GH90, Wad90b, Wad90a]. Unfortunately, these methods do not achieve all the effects of true dynamic data. For instance, dynamic data may be "shared", i.e., embedded in data structures and accessed via different access paths. When shared dynamic data are updated using assignments, the change is visible to all program points that have access to the data. In contrast, when state is being manipulated explicitly, updating shared data involves constructing a fresh copy of the entire data structure in which the data are embedded, and explicitly passing the copy to all program points that need access to the data. This tends to be tedious and error-prone, and results in poor modularity. One particularly faces this difficulty while encoding graph traversal algorithms such as topological sort, unification and the graph reduction execution model of lazy functional languages.

In this paper, we propose a theoretical framework for extending functional languages with dynamic data and assignments while retaining the desirable properties of static values. The resulting language has the following key properties:

- Expressions have static values.
  State-dependent and state-independent expressions are distinguished via a type system. The former are viewed as functions from states to values and the functions themselves are static. (Such functions are called *observers* and resemble classical continuations [Sto77, SW74]). The type system ensures that this view can be consistently maintained, and limits the interaction between observers in such a way that expressions do not have side-effects [1].

- The language is a strict extension of lambda calculus.
  Function abstraction and application have precisely the same meaning as in

[1] In the contemporary functional programming community, the terms "assignment" and "side-effect" are sometimes used synonymously. We use the term "side-effect" in its original meaning: an expression has a side-effect if, in addition to yielding a value, it changes the state in a manner that affects the values of other expressions in the context. Assignments in our proposed language do not have such side-effects. Similar comments apply to terms like "procedure" and "object".

lambda calculus. This is a key property that is not respected by call-by-value languages like Scheme (even in the absence of side-effects). The operational semantics is presented as a reduction system that consists of the standard reduction rules of lambda calculus together with a set of additional rules; these rules exhibit symmetries similar to those of lambda calculus. The reduction system is *confluent* (or, equivalently, Church-Rosser), and recursion-free terms are *strongly normalizing*.

This work can also be given a logical interpretation: it extends the correspondence between logic and programming to include dynamic data. Entities that describe dynamic data, namely references, play a role similar to that of variables in conventional logic. The language described in this paper is the language of constructions for a suitably formulated constructive logic. This dual aspect is treated elsewhere [SR91, Swa91].

The utility of the language is characterized by the following properties:

- Shared dynamic data are available.
  Dynamic data are represented by typed objects called *references*. References can refer to other references as well as to functions. They can be embedded in data structures and used as inputs and outputs of functions.
- Dynamic data may be implemented by a store.
  This is achieved by a type system that sequentializes access to regions of the state, much as in effect systems [GL86] and the languages based on linear logic [GH90, Wad90b, Wad91].
- The language is higher-order.
  References, data structures, functions and observers are all permissible as arguments and results of functions. This permits, for instance, the definition of new control structures, new storage allocation mechanisms, and an object-oriented style of programming.
- The language is integrated symmetrically.
  The applicative sublanguage and the imperative sublanguage are equally powerful and they embed each other. Not only can applicative terms be embedded in imperative terms, but imperative terms can also be embedded in applicative terms. This allows the definition of functions that create and use state internally but are state-independent externally.

The remainder of this paper is organized as follows. Section 2 presents a core formal language called Imperative Lambda Calculus (ILC) that is an extension of the typed lambda calculus. Section 3 studies ILC's use in programming. Section 4 discusses the motivation and design issues behind ILC's type system. Section 5 presents the formal semantics of ILC. This includes a typed denotational semantics and an operational semantics presented as reduction rules. Various formal properties are established, such as type soundness, confluence and strong normalization. Section 6 demonstrates the utility of ILC with an extended example: the unification of first-order terms. Finally, Section 7 compares ILC with related work in the literature.

2 Imperative Lambda Calculus (ILC)

Imperative Lambda Calculus (ILC) is an abstract formal language obtained by extending the typed lambda calculus [Mit90] with imperative programming features. Its main property is that, in spite of this extension, its applicative sublanguage has the same semantic properties as the typed lambda calculus (eg. confluence and strong normalization). Furthermore, these same properties also hold for the entire language of ILC.

2.1 Types

Let β represent the primitive types of ILC. These may include the natural numbers, characters, strings etc. The syntax of ILC types is as follows:

$$
\begin{array}{lcll}
\tau & ::= & \beta \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 & \text{(Applicative types)} \\
\theta & ::= & \tau \mid \mathtt{Ref}\ \theta \mid \theta_1 \times \theta_2 \mid \theta_1 \to \theta_2 & \text{(Mutable types)} \\
\omega & ::= & \theta \mid \mathtt{Obs}\ \tau \mid \omega_1 \times \omega_2 \mid \omega_1 \to \omega_2 & \text{(Observer types)}
\end{array}
$$

The type system is stratified into three layers. The *applicative* layer τ contains the types of the simply typed lambda calculus (extended with pairs). These applicative types include the primitive types β and are closed under product and function space constructions. Note that we use the term "applicative" to refer to the *classical* values manipulated in lambda calculus; semantically, all three layers of ILC are applicative.

The *mutable* layer θ extends the applicative layer with objects called *references*. References are typed values that refer (i.e. point) to values of a particular type. ($\mathtt{Ref}\ \theta$) denotes the type of references that refer to values of type θ. References are used to construct a mutable world (called a *store*) that is used for imperative programming. The world itself is mutable and goes through *states*. The mutable layer includes all applicative types and is closed under the type constructors $\times, \to$ and $\mathtt{Ref}$. Note that references can point to other references, thereby permitting linked data structures. Tuples of references denote mutable records, while reference-returning functions denote mutable arrays.

Finally, the world of the mutable layer needs to be manipulated. In ILC, we take the position that the only manipulation needed for states is *observation* (i.e. inspection). Consider the fact that in the typed lambda calculus, environments are implicitly extended and observed (via the use of variables), but are never explicitly manipulated. Similarly, in ILC, states are implicitly extended and observed (via the use of references), but are never explicitly manipulated. Thus, in a sense, *the world exists only to be observed.* A state differs from an environment in that it may be mutated while being observed; the mutation is restricted to the observation and is not visible to expressions outside the observation.

Observation of the state is accommodated in the *observer* layer ω. This layer includes all applicative and mutable types. In addition, it includes a new type constructor denoted "$\mathtt{Obs}\ \tau$". A value of type $\mathtt{Obs}\ \tau$ is called an *observer*. Such a value observes (i.e. views or inspects) a state and returns a value of type τ. It is significant that the value returned in this fashion is of an applicative type τ. Since

a state exists only to be observed, all information about the state is lost when its observation is completed. So, the values observed in this fashion should be meaningful independent of the state, i.e., they should be applicative. An observer type Obs τ may be viewed as an implicit function space from the set of states to the type τ.

The three layers can be characterized as *kinds* and given category-theoretic semantics. The product and function space constructions have the same meaning in all three layers (cf. Section 5). Thus, there is no ambiguity involved in treating τ types as also being θ and ω types. The name "Imperative Lambda Calculus" is justified by the property that the semantics of functions in all three layers is the same as that of lambda calculus.

2.2 Terms

The abstract syntax of unchecked "preterms" is as follows:

$$\begin{array}{rcl} e & ::= & k \mid x \mid v^* \mid \lambda x{:}\omega.e \mid f(e) \mid (e_1, e_2) \mid e.1 \mid e.2 \\ & & \mid \texttt{letref } v^*{:}\texttt{Ref } \theta := e \texttt{ in } t \\ & & \mid \texttt{get } x{:}\theta \Leftarrow l \texttt{ in } t \\ & & \mid l := e \ ; \ t \end{array}$$

where k are constants, x, v^* are variables, e, e_1, e_2, f, l, t are terms and ω, θ are types.

The constants of* ILC *are limited to be of applicative type. Permissible constants include numbers, booleans, characters and primitive functions on these values. No imperative constants (i.e. no constants involving mutable or observer types) are permitted. This permits us to carefully control the creation and use of the state. We (partially) relax this restriction in section 5.1.

The terms of ILC use two countable sets of variables: ***conventional variables*** and ***reference variables***. Conventional variables are the usual variables of the typed lambda calculus. Reference variables are a new set of variables that share all the properties of conventional variables. Further, distinct reference variables within a term always denote distinct references. References are always introduced by binding them to reference variables; conventional variables can then be bound to such references. This property permits us to reason about the equality of references without recourse to reference constants (which are absent from the language). In the formal presentation, we use an asterisk superscript to distinguish reference variables u^*, v^*, w^* from conventional variables x, y, z. Since the context of a variable determines whether it is a reference or conventional variable, we do not use asterisk superscripts in any of our examples.

Figure 1 presents the context sensitive type syntax of ILC terms. The syntax is expressed as inference rules for judgements of the form $(\Gamma \vdash e{:}\pi)$, where e is a term, π is a type (of any kind τ, θ or ω), and Γ is a sequence of typing assumptions of the forms $(x{:}\pi)$ or $(v^*{:}\texttt{Ref } \theta)$. Γ contains typing assumptions for all the free variables in e.

ILC includes the simply-typed lambda calculus extended with pairs. These terms have their usual meaning in all three layers (rules $\rightarrow$-*intro*, $\rightarrow$-*elim*, $\times$-*intro*,

Constant
$\Gamma \vdash k{:}\tau$
(if k is a constant of type τ)

Weakening
$$\frac{\Gamma \vdash e{:}\pi}{\Gamma, x{:}\pi' \vdash e{:}\pi}$$

Variable hypothesis
$\Gamma, x{:}\pi, \Gamma' \vdash x{:}\pi$

Reference hypothesis
$\Gamma, v^*{:}\texttt{Ref } \theta, \Gamma' \vdash v^*{:}\texttt{Ref } \theta$

$\rightarrow$-*intro*
$$\frac{\Gamma, x{:}\pi_1 \vdash e{:}\pi_2}{\Gamma \vdash (\lambda x{:}\pi_1.e){:}\pi_1 \rightarrow \pi_2}$$

$\rightarrow$-*elim*
$$\frac{\Gamma \vdash f{:}\pi_1 \rightarrow \pi_2 \quad \Gamma \vdash e{:}\pi_1}{\Gamma \vdash f(e){:}\pi_2}$$

$\times$-*intro*
$$\frac{\Gamma \vdash e_1{:}\pi_1 \quad \Gamma \vdash e_2{:}\pi_2}{\Gamma \vdash (e_1, e_2){:}\pi_1 \times \pi_2}$$

$\times$-*elim*
$$\frac{\Gamma \vdash e{:}\pi_1 \times \pi_2}{\Gamma \vdash e.i{:}\pi_i} \quad \text{for } i = 1, 2$$

Obs-intro
$$\frac{\Gamma \vdash t{:}\tau}{\Gamma \vdash t{:}\texttt{Obs } \tau}$$

Obs-elim
$$\frac{\Gamma \vdash t{:}\texttt{Obs } \tau}{\Gamma \vdash t{:}\tau} \quad \text{(if } \Gamma \text{ has only } \tau \text{ types)}$$

Creation
$$\frac{\Gamma, v^*{:}\texttt{Ref } \theta \vdash e{:}\theta \quad \Gamma, v^*{:}\texttt{Ref } \theta \vdash t{:}\texttt{Obs } \tau}{\Gamma \vdash (\texttt{letref } v^*{:}\texttt{Ref } \theta := e \texttt{ in } t){:}\texttt{Obs } \tau}$$

Dereference
$$\frac{\Gamma \vdash l{:}\texttt{Ref } \theta \quad \Gamma, x{:}\theta \vdash t{:}\texttt{Obs } \tau}{\Gamma \vdash (\texttt{get } x{:}\theta \Leftarrow l \texttt{ in } t){:}\texttt{Obs } \tau}$$

Assignment
$$\frac{\Gamma \vdash l{:}\texttt{Ref } \theta \quad \Gamma \vdash e{:}\theta \quad \Gamma \vdash t{:}\texttt{Obs } \tau}{\Gamma \vdash (l := e \ ; \ t){:}\texttt{Obs } \tau}$$

Figure 1: Type inference rules

and $\times$-*elim*). In addition, ILC contains three new observer terms to create a new reference (*creation*), access a reference's content (*dereference*), and modify a reference's content (*assignment*). We now discuss these terms in more detail.

We have seen that there are no reference constants in the language. All references have to be explicitly allocated and bound to a reference variable. This is done by the `letref` construct:

$$\texttt{letref } v^{\bullet}\texttt{:Ref } \theta := e \texttt{ in } t$$

Such a term is an observer of the same type as t (rule *Creation*). When used to observe a state, it extends the state by creating a new reference, extends the environment by binding $v^{\bullet}$ to the reference, initializes the reference to the value of e in the extended environment, and finally observes the value of t in the extended environment and state.

The mutable world of references may be inspected by dereferencing a reference, i.e. by inspecting the value that the reference points to, or, using alternate terminology, by inspecting the reference's *content*. If l is a reference-valued expression of type `Ref` θ, then a term of the form

$$\texttt{get } x\texttt{:}\theta \Leftarrow l \texttt{ in } t \ .$$

binds x to the content of l, and denotes the value of t in the extended environment. Here, t must be an observer of type `Obs` τ, and the entire term is again an observer of type `Obs` τ (rule *Dereference*).

Finally, the content of a reference may be modified via assignment observers of the form

$$l := e \ ; \ t$$

where l is of type `Ref` θ and e is of type θ, for some θ (rule *Assignment*). When used to observe a state, an assignment observer modifies the reference l to refer to e, and observes the value of t in the modified state. Note that "$l := e$" is not a term by itself as in conventional languages. The state is modified *for* the observer t, and the entire construct is again an observer.

The lifetime of a mutable world (i.e. a collection of references) is limited to its observation. So, the creation of $v^{\bullet}$ and the modification of l are observable only within the bodies t of the creation and assignment observers respectively, and there are no side effects produced by the observers. If there are no free occurrences of reference variables or other state-dependent variables in an observer term, then the term is a trivial observer that is independent of any state. Such an observer can be coerced to an applicative term (rule *Obs-elim*). Conversely, every applicative term (every term of a τ type) is trivially an observer (rule *Obs-intro*).

It is important to note that all the primitive constructions on observers (get, letref and assignment) involve exactly one subterm of an observer type. This reflects the requirement that manipulations of state should be performed in a sequential fashion, similar in spirit to the proposal of single-threaded lambda calculus [GH90]. Even though it is possible to express functions which accept more than one observer, the state manipulations of such observers have to be eventually sequentialized because there are no multi-ary primitives on observers. (Recall that there are no constants of mutable and observer types). This fact has two

consequences. First, programming in the imperative sublanguage of ILC requires a continuation-passing style. Second, the state can be implemented efficiently by means of a global store. We return to these issues in section 4.

3 ILC as a Programming Language

ILC can be used as a programming language in different styles. It can be used as a purely applicative language by restricting oneself to applicative types. It can be used as a purely imperative language by mainly using observers (this requires a continuation-passing style of programming). These styles correspond to traditional programming paradigms.

ILC also permits an interesting new style of programming. It permits closed imperative observers to be embedded in applicative terms (via the rule *Obs-elim*). Applicative terms can be freely embedded in imperative observers (via the rule *Obs-intro*). Higher-order functions and laziness can be used to glue together both imperative and applicative subcomputations, though imperative computation is restricted to continuation-passing style.

One extreme of this paradigm is to use ILC with imperative observers at the top level, but with nontrivial applicative subcomputations involving higher-order functions. This use is similar to that of Haskell where state-oriented input/output operations are usually carried out at the top level. More generally, ILC can be used with imperative observers embedded in applicative expressions (via the rule *Obs-elim*). This corresponds to the use of side-effect-free function procedures in Algol-like languages.

The examples in this paper further illustrate this style of programming. Example 1 (factorial) displays how imperative computations can be embedded in applicative terms. Example 2 (a movable point object) exhibits how imperative computations can be encapsulated as closures. The unification example of Section 6 demonstrates how the laziness of observers permits them to be passed to functions and returned as results.

3.1 Syntactic sugar for dereferencing

The need to use get's for dereferencing is rather tedious: it forces us to choose new names, and more importantly, it clutters up the code. The tedium can be alleviated to a large extent through a simple notational mechanism.

Abbreviation: If $(\texttt{get } x \Leftarrow l \texttt{ in } t[x])$ is an observer term with no occurrence of x in a proper observer subterm of t, then we allow it to be abbreviated as $t[l\uparrow]$. $l\uparrow$ may be read informally as "the current content of reference l".

Expansion: If t is an observer term with a particular occurrence of $l\uparrow$, then it is expanded by introducing a "get" at the smallest observer subterm of t containing the occurrence of $l\uparrow$.

The intuition behind this abbreviation is that an observer term $(\texttt{get } x \Leftarrow l \texttt{ in } t)$ is a program point at which the content of l is observed, while occurrences of x in

l are program points at which that content is used. If l is never modified between the point of dereference and a point of use, then we can safely view the dereference as taking place at the point of use. For example,

$$(p := n\uparrow\ ; *p\uparrow\ ;\ n := n\uparrow -1;\ c)$$
$$= \mathbf{get}\ x \Leftarrow n\ \mathbf{in}\ \mathbf{get}\ y \Leftarrow p\ \mathbf{in}$$
$$p := x * y;$$
$$\mathbf{get}\ z \Leftarrow n\ \mathbf{in}$$
$$n := z - 1;\ c$$

$$l\uparrow\uparrow = (\mathbf{get}\ x \Leftarrow l\ \mathbf{in}\ x\uparrow) = (\mathbf{get}\ x \Leftarrow l\ \mathbf{in}\ \mathbf{get}\ y \Leftarrow x\ \mathbf{in}\ y)$$

$$f(l\uparrow) = (\mathbf{get}\ x \Leftarrow l\ \mathbf{in}\ f(x)) \qquad \text{if } f : \tau_1 \to \mathbf{Obs}\ \tau_2$$

$$f(l\uparrow) = f(\mathbf{get}\ x \Leftarrow l\ \mathbf{in}\ x) \qquad \text{if } f : \mathbf{Obs}\ \tau_1 \to \mathbf{Obs}\ \tau_2$$

3.2 Examples

For our examples, we assume that ILC is enhanced with user-defined type constructors and record types (drawn from standard ML [MTH90]). We also assume that ILC is enhanced with *explicit* parametric polymorphism with types ranging over the universe of applicative (τ) types. Implicit polymorphism is problematic in the presence of references and assignments [Tof88] — explicit polymorphism does not suffer from these problems. In our examples, we erase explicit type quantification and type application, and leave it to the reader to fill in the missing information.

We also assume primitives such as `case`, `let`, `letrec` and `if-then-else` for all types. Note that these primitives violate our earlier prohibition of primitives over mutable and observer types. In section 5.1, we shall see that such primitives are indeed permissible.

Example 1: Factorial

This trivial example is meant to provide an initial feel for the language, and illustrate how imperative observers can be embedded in applicative expressions. This example is not meant to illustrate the benefits of ILC; indeed, a preferred solution is to write this as a tail-recursive applicative function and have the compiler optimize the code into an iterative loop.

```
factorial = λm:nat. letref n:Ref nat := m in
                    letref acc:Ref nat := 1 in
                    letrec fact:Obs nat = if (n↑< 2) then acc↑
                                          else acc := n↑*acc↑;
                                               n := n↑-1;
                                               fact
                    in fact
```

The function `factorial` has no free references or state-dependent variables, and so has the applicative type (`nat` → `nat`). This means that `factorial` can be freely embedded in applicative expressions even though it contains imperative subcomputations.

Example 2: Points

We implement a point object that hides its internal state and exports operations. Let `Point` be the type of objects that represent movable planar points.

$$\begin{aligned}\mathtt{Point} = \{&\mathtt{x_coord} : (\mathbf{Real} \to \mathbf{Obs}\ T) \to \mathbf{Obs}\ T,\\ &\mathtt{y_coord} : (\mathbf{Real} \to \mathbf{Obs}\ T) \to \mathbf{Obs}\ T,\\ &\mathtt{move} : (\mathbf{Real} \times \mathbf{Real}) \to \mathbf{Obs}\ T \to \mathbf{Obs}\ T,\\ &\mathtt{equal} : \mathtt{Point} \to (\mathbf{Bool} \to \mathbf{Obs}\ T) \to \mathbf{Obs}\ T\}\end{aligned}$$

The function `mkpoint` implements objects of type `Point`.

```
mkpoint : (Real → Real → (Point → Obs T) → Obs T)
        = λx. λy. λk.
          letref xc:Real := x in
          letref yc:Real := y in
          k({x_coord = λk. k(xc↑),
             y_coord = λk. k(yc↑),
             move = λ(dx,dy). λc. xc := xc↑+dx; yc := yc↑+dy; c,
             equal = λ{x_coord,y_coord,move,equal}. λk.
                       x_coord(λx. y_coord(λy.
                           k(x = xc↑ and y = yc↑)))
          })
```

Note, first of all, that the `mkpoint` operation cannot simply yield a value of type `Point` because it is not an applicative value. The extent of xc and yc is limited to the bodies of the `letref`s which allocate these references; hence the entire computation which uses these references must occur in these bodies. Therefore, `mkpoint` is defined to accept a point observer function k and pass it the newly created point. This is similar to the continuation-passing style of programming. Observers here play the role of continuations.[2] Such continuation-passing style functions can be defined more conveniently using Wadler's monad comprehension notation [Wad90a]. Note that each operation in the object is similarly defined in the continuation-passing style.

This example demonstrates that state-encapsulating closures are available in ILC, albeit in the continuation-passing style. Such closures are also representable in semi-functional languages like Scheme and Standard ML, but usually involve side-effects.

[2] Technically speaking, observers are not continuations because they return values. But, they can be thought of as continuations in the imperative sublanguage so that the "answers" produced can then be consumed in the applicative sublanguage.

4 Discussion of ILC

The motivation behind ILC's type system is threefold. First, we wish to exclude imperative terms that "export" their local effects. Consider the unchecked preterm (`letref` $v := 0$ `in` v). This term, if well-typed, would export the locally created reference v outside its scope resulting in a dangling pointer. Closures that capture references are prohibited for the same reason — they export state information beyond its local scope. The type system prohibits such terms by requiring the value returned by an observer to be applicative and hence free of state information. (Recall that observer types are of the form `Obs` τ where τ is an applicative type).

Second, we wish to ensure that the imperative sublanguage can be implemented efficiently without causing side-effects. Consider the unchecked preterm

$$v := 0\,;\ ((v := 2\,;\ \texttt{get}\ x \Leftarrow v\ \texttt{in}\ x) + (\texttt{get}\ x \Leftarrow v\ \texttt{in}\ x))$$

In a language with a global store and global assignments (eg. ML or Scheme), the value of the term depends on the order of evaluation of +'s arguments. Further, the term has the side-effect of changing the value of the global reference v to 2. On the other hand, if assignments are interpreted to have local effects, then the value of the term would be 2 regardless of the order of evaluation, and the term would not have any side-effects. However, the state can no longer be implemented by a (global) store. The state needs to be copied and passed to each argument of +, making the language quite inefficient.

The type system of ILC excludes such terms from the language by requiring that all state-manipulations be performed in a sequential fashion. Well-typed terms of ILC do not require the state to be copied, and hence the state can be implemented by a (global) store. The only legal way to express the above example in ILC, is to sequentialize its assignments. For example,

$$v := 0\,;\ \texttt{get}\ x \Leftarrow v\ \texttt{in}\ (v := 2\,;\ \texttt{get}\ y \Leftarrow v\ \texttt{in}\ (y + x))$$

is a well-typed term.

ILC distinguishes between state-dependent observers and applicative values. Both observers and values can be passed to functions and returned as results — it is not necessary to evaluate an observer to a value before passing it. In fact, an observer of the form (`get` $x \Leftarrow l$ `in` t) is in head normal form, just as a lambda expression is in head normal form (see section 5.2). This is a form of laziness and, in fact, directly corresponds to Algol's call by name. However, an observer passed to a function can only be evaluated in a single state due to the single-threaded nature of the type system. So, the ambiguities caused by Algol's call-by-name are not shared by ILC.

Finally, we wish the type system to ensure that all recursion-free terms are strongly normalizable, i.e., their evaluation always terminates. We postpone a discussion of this issue to section 5.2. For now, we merely note that strong normalization is achieved by making observers non-storable values. If strong normalization is not considered critical, the θ and ω layers may be conflated.

The type system described thus far is overly restrictive. It prohibits all nonsequential combinations of observers in order to ensure that the state is never copied.

For example, a term of the form

$$(v := 0\,;\ (\texttt{get}\ x \Leftarrow v\ \texttt{in}\ x) + (\texttt{get}\ x \Leftarrow v\ \texttt{in}\ x))$$

is excluded because the observer arguments of + are combined nonsequentially. However, this term does not require the state to be copied since the arguments of + do not locally modify the state. This suggests that the type system could be relaxed by distinguishing between:

- ***creators***, that locally extend the state;
- ***pure observers***, that observe the state without locally extending or modifying it; and
- ***mutators***, that locally modify the state.

The type system could then permit certain state-dependent terms to be combined. For example, two pure observers could be safely combined. To be effective, such a solution would also have to incorporate a comprehensive type system that captures "effects" of expressions on "regions" of references (similar to that of FX [LG88]). This would permit combining mutators that mutate disjoint regions of the state. We do not explore this solution in this paper because it is orthogonal to the issues considered here. It is also clear that such a type system does not completely eliminate the need for sequencing.

5 Semantics of ILC

We present the denotational and operational semantics of ILC, and sketch the proofs of several important properties including soundness, strong normalization and confluence.

5.1 Denotational semantics

The denotational semantics is defined using complete partial orders (cpo's) as domains. For every primitive type β, choose a domain D_β. $D_{\tau\times\tau}$ and $D_{\tau\to\tau}$ are defined by the standard product and continuous function space constructions on cpo's.

For every reference type `Ref` θ, choose a countable flat domain $D_{\texttt{Ref}\ \theta}$. The defined elements of a $D_{\texttt{Ref}\ \theta}$ domain should be disjoint from those of any other such domain. The defined elements of these domains may be thought of as "locations". `State` is the set of partial mappings σ from $\cup_\theta D_{\texttt{Ref}\ \theta}$ to $\cup_\theta D_\theta$ with the constraint that, whenever $\alpha \in D_{\texttt{Ref}\ \theta}$, $\sigma(\alpha) \in D_\theta$ and $\sigma(\perp_{\texttt{Ref}\ \theta}) = \perp_\theta$. The subset of $\cup_\theta D_{\texttt{Ref}\ \theta}$ mapped by σ is denoted $dom(\sigma)$. σ_0 is the "empty" state, i.e., $dom(\sigma_0)$ contains only $\perp_{\texttt{Ref}\ \theta}$ elements.

The domain for an observation type is $D_{\texttt{Obs}\ \tau} = [\texttt{State} \to D_\tau]$.

An environment η is a mapping from variables to $\cup_\omega D_\omega$. If Γ is a type assignment, we say that η satisfies Γ if $\eta(x) \in D_\omega$ for every $x{:}\omega \in \Gamma$, $\eta(v^\circ) \in D_{\texttt{Ref}\ \theta}$ for every $v^\circ{:}\texttt{Ref}\ \theta \in \Gamma$, and $\eta(v^\circ) \neq \eta(w^\circ)$ for every $v^\circ, w^\circ{:}\texttt{Ref}\ \theta \in \Gamma$.

$$
\begin{array}{lcl}
[\![\Gamma \vdash x:\omega]\!]\,\eta & = & \eta x \\
[\![\Gamma \vdash (\lambda x{:}\omega_1.e){:}\omega_1 \to \omega_2]\!]\,\eta & = & \lambda v \in D_{\omega_1}.[\![\Gamma, x{:}\omega_1 \vdash e{:}\omega_2]\!]\,(\eta[x \to v]) \\
[\![\Gamma \vdash f(e){:}\omega_2]\!]\,\eta & = & ([\![\Gamma \vdash f{:}\omega_1 \to \omega_2]\!]\,\eta)([\![\Gamma \vdash e{:}\omega_1]\!]\,\eta) \\
[\![\Gamma \vdash (e_1,e_2){:}\omega_1 \times \omega_2]\!]\,\eta & = & \langle [\![\Gamma \vdash e_1{:}\omega_1]\!]\,\eta, [\![\Gamma \vdash e_2{:}\omega_2]\!]\,\eta\rangle \\
[\![\Gamma \vdash e.1{:}\omega_1]\!]\,\eta & = & \mathtt{fst}([\![\Gamma \vdash e{:}\omega_1 \times \omega_2]\!]\,\eta) \\
[\![\Gamma \vdash e.2{:}\omega_2]\!]\,\eta & = & \mathtt{snd}([\![\Gamma \vdash e{:}\omega_1 \times \omega_2]\!]\,\eta)
\end{array}
$$

$$
\begin{array}{lcl}
[\![\Gamma \vdash e{:}\tau]\!]\,\eta & = & [\![\Gamma \vdash e{:}\mathtt{Obs}\ \tau]\!]\,\eta\,\sigma_\theta \\
[\![\Gamma \vdash e{:}\mathtt{Obs}\ \tau]\!]\,\eta & = & \lambda\sigma.[\![\Gamma \vdash e{:}\tau]\!]\,\eta
\end{array}
$$

$$
\begin{array}{l}
[\![\Gamma \vdash (\mathtt{letref}\ v^\circ{:}\mathtt{Ref}\ \theta := e\ \mathtt{in}\ t){:}\mathtt{Obs}\ \tau]\!]\,\eta \\
\qquad = \lambda\sigma.[\![\Gamma, v^\circ{:}\mathtt{Ref}\ \theta \vdash t{:}\mathtt{Obs}\ \tau]\!]\,(\eta[v^\circ \to \alpha])\,(\sigma[\alpha \to v_e]) \\
\qquad\quad \text{where } \alpha \text{ is any element of } D_{\mathtt{Ref}\ \theta} \text{ not in } dom(\sigma) \\
\qquad\quad \text{and } v_e = [\![\Gamma, v^\circ{:}\mathtt{Ref}\ \theta \vdash e{:}\theta]\!]\,(\eta[v^\circ \to \alpha]) \\
[\![\Gamma \vdash (\mathtt{get}\ x{:}\theta \Leftarrow l\ \mathtt{in}\ t){:}\mathtt{Obs}\ \tau]\!]\,\eta \\
\qquad = \lambda\sigma.[\![\Gamma, x{:}\theta \vdash t{:}\mathtt{Obs}\ \tau]\!](\eta[x \to \sigma([\![\Gamma \vdash l{:}\mathtt{Ref}\ \theta]\!]\eta)])\,\sigma \\
[\![\Gamma \vdash (l := e; t){:}\mathtt{Obs}\ \tau]\!]\,\eta \ = \ \lambda\sigma.[\![\Gamma \vdash t{:}\mathtt{Obs}\ \tau]\!]\,\eta\,(\sigma[v_l \to v_e]) \\
\qquad\quad \text{where } v_l = [\![\Gamma \vdash l{:}\mathtt{Ref}\ \theta]\!]\eta \text{ and } v_e = [\![\Gamma \vdash e{:}\theta]\!]\eta
\end{array}
$$

Figure 2: Denotational semantics

The denotational semantics of ILC (see figure 2) is defined by induction on type derivations. The meaning of an expression ($\Gamma \vdash e{:}\omega$) is a mapping from environments satisfying Γ to D_ω. (See [Mit90] for a discussion of this notation).

Lemma 1 *$[\![\Gamma \vdash e{:}\omega]\!]$ is well-defined.*

This involves showing that continuous functions in the interpretation of λ are unique and that the choice of α in the interpretation of `letref` is immaterial.

Proposition 2 *$[\![\Gamma \vdash e{:}\omega]\!]\eta \in D_\omega$ whenever η satisfies Γ.*

This is proved by a simple induction on type derivations. The main property to be verified is that η and σ are always extended or modified in a manner type-consistent with Γ. We present the proof case for `letref` terms; other cases can be verified similarly. Assume that η satisfies Γ.

- $[\![\Gamma \vdash (\mathtt{letref}\ v^\circ{:}\mathtt{Ref}\ \theta := e\ \mathtt{in}\ t){:}\mathtt{Obs}\ \tau]\!]\,\eta \in D_{\mathtt{Obs}\ \tau}$

  Let $\alpha \in D_{\mathtt{Ref}\ \theta}$. Then, since η satisfies Γ, $(\eta[v^\circ \to \alpha])$ satisfies $\Gamma, v^\circ{:}\mathtt{Ref}\ \theta$. Thus, by induction hypothesis, $[\![\Gamma, v^\circ{:}\mathtt{Ref}\ \theta \vdash t{:}\mathtt{Obs}\ \tau]\!]\,(\eta[v^\circ \to \alpha]) \in D_{\mathtt{Obs}\ \tau}$ and $v_e = [\![\Gamma, v^\circ{:}\mathtt{Ref}\ \theta \vdash e{:}\theta]\!]\,(\eta[v^\circ \to \alpha]) \in D_\theta$. Thus, $(\sigma[\alpha \to v_e])$ is a well-formed state, and hence $\lambda\sigma.[\![\Gamma, v^\circ{:}\mathtt{Ref}\ \theta \vdash t{:}\mathtt{Obs}\ \tau]\!]\,(\eta[v^\circ \to \alpha])\,(\sigma[\alpha \to v_e]) \in D_{\mathtt{Obs}\ \tau}$ as desired.

This property ensures that every expression of an applicative type τ is free of state information. It also shows that observers (of type `Obs` τ) do not have any visible side-effects. This proves our claim that ILC is free of side-effects.

We note that the semantics uses the state in a single-threaded fashion [Sch85]. Whenever a state is updated, the old state is discarded. Thus, the semantics can indeed be realized by a global store and no side effects need enter the implementation through the "back door".

At this stage, we can also point out what kind of primitive constants of mutable and observer types may be added to the language without violating the basic framework. The acceptable constants should be purely *combinatorial*, i.e., they should not use any information about the semantic interpretations of their parameters. For example, the constants $if_{\mathtt{Obs}\ \tau}{:}\mathtt{Bool} \times \mathtt{Obs}\ \tau \times \mathtt{Obs}\ \tau \to \mathtt{Obs}\ \tau$ defined by

$$if_{\mathtt{Obs}\ \tau}(p, t_1, t_2) = \begin{cases} t_1, & \text{if } p = true \\ t_2, & \text{if } p = false \end{cases}$$

are acceptable because they are not dependent on the semantic interpretation of the `Obs` τ parameters. On the other hand, the constant $add{:}\mathtt{Obs}\ \tau \times \mathtt{Obs}\ \tau \to \mathtt{Obs}\ \tau$ defined by

$$add(t_1, t_2) = \lambda\sigma.\,(t_1\sigma + t_2\sigma)$$

is not acceptable as it interprets `Obs` τ parameters to be functions of type $[\mathtt{State} \to D_\tau]$.

5.2 Reduction semantics

We now present reduction rules for terms of ILC. These rules are meant to reduce terms to normal form such that every closed term of a primitive type β reduces to a constant of that type. Let $V(t)$ be the set of free variables of term t.

The reduction rules presented in figure 3 propagate `get` terms outward until they encounter a `letref` or assignment, and then discharge the `get` construct. The `letref` and assignment constructs can be discharged only after the state observation in their body is completed, i.e., after the body reduces to applicative term. At that stage, the body would be a "value term" of the form k, $\lambda x{:}\omega.e_1$ or (e_1, e_2). Rules (3) and (6) handle this situation.

Let $\xrightarrow{*}$ be the reflexive, transitive closure of $\longrightarrow$. We can easily show that one step reduction preserves types, and by induction, so does $\xrightarrow{*}$.

Lemma 3 (Type preservation) *If $(\Gamma \vdash s{:}\omega)$ is a term, and $s \longrightarrow t$, then $(\Gamma \vdash t{:}\omega)$ is a term.*

We can also show that one step reduction preserves meaning.

Proposition 4 (Soundness) *Let $(\Gamma \vdash s{:}\omega)$ and $(\Gamma \vdash t{:}\omega)$ be terms, and let $s \longrightarrow t$. Then*

$$[\![\Gamma \vdash s{:}\omega]\!]\,\eta \equiv [\![\Gamma \vdash t{:}\omega]\!]\,\eta$$

Proof: Rules (1) and (2) are classical. For (3) and (6), note that if u is of the stated form, it can only be a trivial observer which is also of an applicative type τ. (Nontrivial observers have a `letref`, `get` or ":=" at their principal position.)

Let $u ::= k \mid \lambda x{:}\omega.e_1 \mid (e_1, e_2)$

(1)	$(\lambda x{:}\omega.e_1)(e_2)$	$\longrightarrow$	$e_1[e_2/x]$	
(2)	$(e_1, e_2).i$	$\longrightarrow$	e_i	for $i = 1, 2$
(3)	$\texttt{letref } v^\circ{:}\texttt{Ref } \theta := e \texttt{ in } u$	$\longrightarrow$	u	if $v^\circ \notin V(u)$
(4)	$\texttt{letref } v^\circ{:}\texttt{Ref } \theta := e \texttt{ in}$ $\texttt{get } x{:}\theta' \Leftarrow v^\circ \texttt{ in } t$	$\longrightarrow$	$\texttt{letref } v^\circ{:}\texttt{Ref } \theta := e$ $\texttt{in } t[e/x]$	
(5)	$\texttt{letref } v^\circ{:}\texttt{Ref } \theta := e \texttt{ in}$ $\texttt{get } x{:}\theta' \Leftarrow w^\circ \texttt{ in } t$	$\longrightarrow$	$\texttt{get } x'{:}\theta' \Leftarrow w^\circ \texttt{ in}$ $\texttt{letref } v^\circ{:}\texttt{Ref } \theta := e$ $\texttt{in } t[x'/x]$	if $v^\circ \neq w^\circ$ and $x' \notin V(e) \cup V(t)$
(6)	$v^\circ := e\ ;\ u$	$\longrightarrow$	u	
(7)	$v^\circ := e\ ;$ $\texttt{get } x{:}\theta \Leftarrow v^\circ \texttt{ in } t$	$\longrightarrow$	$v^\circ := e\ ;\ t[e/x]$	
(8)	$v^\circ := e\ ;$ $\texttt{get } x{:}\theta \Leftarrow w^\circ \texttt{ in } t$	$\longrightarrow$	$\texttt{get } x'{:}\theta \Leftarrow w^\circ \texttt{ in}$ $v^\circ := e\ ;\ t[x'/x]$	if $v^\circ \neq w^\circ$ and $x' \notin V(e) \cup V(t)$

Figure 3: Reduction rules

Hence, u is state-independent. For example,

$$\begin{aligned}
&[\![\Gamma \vdash (\texttt{letref } v^\circ{:}\texttt{Ref } \theta := e \texttt{ in } u){:}\texttt{Obs } \tau]\!]\,\eta \\
&= \lambda\sigma.[\![\Gamma, v^\circ{:}\texttt{Ref } \theta \vdash u{:}\texttt{Obs } \tau]\!](\eta[v^\circ \to \alpha])(\sigma[\alpha \to v_e]) \\
&\qquad \text{where } \alpha \text{ is any element of } D_{\texttt{Ref } \theta} \text{ not in } dom(\sigma) \\
&\qquad \text{and } v_e = [\![\Gamma, v^\circ{:}\texttt{Ref } \theta \vdash e{:}\theta]\!](\eta[v^\circ \to \alpha]) \\
&= \lambda\sigma.[\![\Gamma, v^\circ{:}\texttt{Ref } \theta \vdash u{:}\tau]\!]\,\eta[v^\circ \to \alpha] \\
&= \lambda\sigma.[\![\Gamma \vdash u{:}\tau]\!]\,\eta \qquad \text{since } v^\circ \notin V(u) \\
&= [\![\Gamma \vdash u{:}\texttt{Obs } \tau]\!]\,\eta
\end{aligned}$$

For (4) and (5), recall that v° and w° are "reference variables" which are only bound in letref constructs. By the denotational semantics, any two such variables denote distinct references unless they are syntactically identical. For example,

$$\begin{aligned}
&[\![\Gamma \vdash (\texttt{letref } v^\circ{:}\texttt{Ref } \theta := e \texttt{ in } (\texttt{get } x{:}\theta \Leftarrow v^\circ \texttt{ in } t)){:}\texttt{Obs } \tau]\!]\,\eta \\
&= \lambda\sigma.[\![\Gamma, v^\circ{:}\texttt{Ref } \theta \vdash (\texttt{get } x{:}\theta \Leftarrow v^\circ \texttt{ in } t){:}\texttt{Obs } \tau]\!](\eta[v^\circ \to \alpha])(\sigma[\alpha \to v_e]) \\
&\qquad \text{where } \alpha \text{ is any element of } D_{\texttt{Ref } \theta} \text{ not in } dom(\sigma) \\
&\qquad \text{and } v_e = [\![\Gamma, v^\circ{:}\texttt{Ref } \theta \vdash e{:}\theta]\!](\eta[v^\circ \to \alpha]) \\
&= \lambda\sigma.[\![\Gamma, v^\circ{:}\texttt{Ref } \theta, x{:}\theta \vdash t{:}\texttt{Obs } \tau]\!](\eta[v^\circ \to \alpha][x \to v_e])(\sigma[\alpha \to v_e]) \\
&= \lambda\sigma.[\![\Gamma, v^\circ{:}\texttt{Ref } \theta \vdash t[e/x]{:}\texttt{Obs } \tau]\!](\eta[v^\circ \to \alpha])(\sigma[\alpha \to v_e]) \\
&= [\![\Gamma \vdash (\texttt{letref } v^\circ{:}\texttt{Ref } \theta := e \texttt{ in } t[e/x]){:}\texttt{Obs } \tau]\!]\,\eta
\end{aligned}$$

Rules (7) and (8) are similar. □

Strong normalization is considered to be a desirable property of typed programming languages. It asserts that the evaluation of a well-typed recursion-free term *always* terminates. Conceptually, its significance is that all terms are meaningful; there are no undefined terms [Pra71]. Its pragmatic implication is that nontermination is limited to explicit recursion. Strong normalization is ensured in ILC by making observers non-storable. If observers were storable, `Ref Obs nat` would be a well-formed type and the language would contain the following infinite reduction sequence:

$$\begin{aligned}
&(\texttt{letref } u^\circ : \texttt{Ref Obs } T := (u^\circ\,\uparrow) \texttt{ in } u^\circ\,\uparrow) \\
&\quad \longrightarrow (\texttt{letref } u^\circ : \texttt{Ref Obs } T := (u^\circ\,\uparrow) \texttt{ in } u^\circ\,\uparrow) \\
&\quad \longrightarrow \ldots
\end{aligned}$$

Since u° : `Ref Obs nat`, we can store in it an observation of itself ($u^\circ\,\uparrow$). Indeed, recursion is defined in Scheme by a similar device [RC86].

Proposition 5 (Strong Normalization) *Let $(\Gamma \vdash t : \omega)$ be a recursion-free term. Then there is no infinite reduction sequence $t \longrightarrow t_1 \longrightarrow t_2 \longrightarrow \ldots$ of well-typed* ILC *terms.*

Proof: The proof of the above proposition is quite elaborate and uses twin induction on types and terms, along the classical lines of [Tai75, GLT89]. The proof may be found in [Swa91].

The Church-Rosser property for the reduction system may be established as follows. Let V be a countably infinite set of reference variables. Treat the reduction rules (4) and (7) as schematic rules representing an infinite set of rules, one for each $v^\circ \in V$. Similarly, the rules (5) and (8) may be treated as being schematic for an infinite set of rules, one for each distinct pair of $v^\circ, w^\circ \in V$. The resulting reduction system has no "critical overlaps", i.e., no left hand side has a common instance with a subterm of another left hand side, unless the subterm is a metavariable. So, it follows that:

Lemma 6 (Local Confluence) *If $(\Gamma \vdash r : \omega)$ is a term, and if $r \longrightarrow s_1$ and $r \longrightarrow s_2$, then there is a term $(\Gamma \vdash t : \omega)$ such that $s_1 \xrightarrow{\circ} t$ and $s_2 \xrightarrow{\circ} t$.*

Hence, by Newman's Lemma, we have

Proposition 7 (Confluence) *If $(\Gamma \vdash r : \omega)$ is a term, and if $r \xrightarrow{\circ} s_1$ and $r \xrightarrow{\circ} s_2$, then there is a term $(\Gamma \vdash t : \omega)$ such that $s_1 \xrightarrow{\circ} t$ and $s_2 \xrightarrow{\circ} t$.*

This result can be extended to the language with recursion as follows. Add the following reduction rule

$$(9) \quad \text{fix } e \longrightarrow e(\text{fix } e)$$

where *fix* is the least fixed point operator for each type ω. The resulting system still has no critical overlaps. Further, it is left-linear, i.e., there are no repeated occurrences of metavariables on any left hand side. Hence, by Huet [Hue80], Lemma 3.3, we have

Proposition 8 *The reduction system (1-9) is confluent.*

This property, which is equivalent to the Church-Rosser property, gives further evidence of the side-effect-freedom of ILC. If there were side effects, then the

evaluation of a subexpression would affect the meaning of its context, and the normal forms would be dependent on the evaluation order.

The independence of results on the evaluation order means, in particular, that call-by-value and call-by-name evaluations produce the same results. This observation must be interpreted carefully. In the lambda calculus setting, the distinction between call-by-value and call-by-name refers to when the arguments to a function are *evaluated*. We are using these terms in the same sense. However, in the imperative programming framework, the terms call-by-value and call-by-name are used to make a different distinction — the question of when the arguments to a function are *observed*. In the terminology of ILC, this involves a coercion from a type Obs τ to a type τ. Since Obs τ represents the function space $[\mathtt{State} \to D_\tau]$, such a coercion involves change of semantics. ILC permits no such coercion. Thus, in the imperative programming sense, ILC's parameter passing is call-by-name. However, the linearity of observer constructions means that a function accepting an observer can use it to observe at most one state. This contrasts with Algol 60, where a call-by-name parameter can be used to observe many states with quite unpredictable effects.

6 Extended Example : Unification

To illustrate the expressive power and usability of the language, we implement unification by an algorithm that performs *shared updates* on a data structure. Unification [Rob65] is a significant problem that finds applications in diverse problems including type inference, implementation of Prolog and theorem provers, and natural semantics.

Figure 4 contains an ILC program that computes the most general common instance of two terms t_1 and t_2. A *term* is either a variable or a pair denoting the application of a function symbol to a list of subterms. A variable is represented by a *reference*. The reference may contain either a term (if the variable is already bound), or the special value Unbound. This representation of terms illustrates the notion of *shared dynamic data* mentioned in Section 1; a function accepting a term has indirect access to all the references embedded in the term.

We assume the existence of a global reference called sigma that accumulates the list of references bound during an attempt at unification. If the unification is successful, this yields the most general unifier, while the representations of the terms t_1 and t_2 correspond to the most general common instance. On failure, this list is used to reset the values of the references to Unbound.

The function unify attempts to compute the most general common instance of two terms t and u. If the unification is successful, it instantiates both t and u to their most general common instance (by updating the references embedded in them), and evaluates the success continuation sc. If the unification is unsuccessful, it leaves the terms unchanged and evaluates the failure continuation fc. Internally, it uses the auxiliary function unify-aux which updates the terms in both cases. By providing the failure continuation (undo fc) to this function, terms are restored to their original values upon failure. The function unify-lists unifies two lists of terms (lt, lu), bind unifies a variable v with a term u, occurs checks whether a variable v occurs free in a term u, and undo resets the values of variables that

```
datatype  term  =  Var of Ref var
                |  Apply of (symbol × List term)

    and   var   =  Unbound | Bound of term

unify: term × term × Obs T × Obs T → Obs T
   = λ(t, u, sc, fc). unify-aux(t, u, sc, undo(fc))

unify-aux: term × term × Obs T × Obs T → Obs T
   = λ(t, u, sc, fc).
       case (t, u) of
         (Var(v1), Var(v1)) ⇒ sc
       | (Var(v1), Apply(f, ts)) ⇒ bind(v1, u, sc, fc)
       | (Apply(f, ts), Var(v2)) ⇒ bind(v2, t, sc, fc)
       | (Apply(f, ts), Apply(g, us)) ⇒
           if (f = g) then unify-lists(ts, us, sc, fc) else fc

unify-lists: List term × List term × Obs T × Obs T → Obs T
   = λ(lt, lu, sc, fc). case (lt, lu) of
                          ([], []) ⇒ sc
                        | (t :: ts, u :: us) ⇒
                            unify-aux(t, u, unify-lists(ts, us, sc, fc), fc)
                        | (_, _) ⇒ fc

bind: Ref var × term × Obs T × Obs T → Obs T
   = λ(v, u, sc, fc). case v↑ of
                        Unbound ⇒ occurs(v, u, fc, (v := Bound(u);
                                                    sigma := v :: sigma↑;
                                                    sc))
                      | Bound(t) ⇒ unify-aux(t, u, sc, fc)

undo: Obs T → Obs T
= λfc. case sigma↑ of
         [] ⇒ fc
       | v :: vs ⇒ v := Unbound;
                   sigma := vs;
                   undo(fc)

occurs: Ref var × term × Obs T × Obs T → Obs T
   = ...
```

Figure 4: Unification of first-order terms

have been bound during a failed attempt at unification. The definition of `occurs` is straightforward and has been omitted.

The significant aspect of this program is that when an unbound variable is unified with a term that does not contain any free occurrence of the variable, unification succeeds by *assigning* the term to the reference that represents the variable (see function `bind`). This modification is visible via other access paths to the reference. It is this information sharing that affects the unification of subsequent subterms, even though no values are *passed* between these program points. In contrast, in a pure functional language, every computed value needs to be passed explicitly to all program points which need it. In the unification example, this means that whenever a variable is modified, the modified value needs to be passed to all other subterms that are yet to be unified, an expensive proposition indeed.

7 Related Work

In this section, we compare our research with related work. We organize this comparison based on the broad approach taken by the related work.

Linearity Substantial research has been devoted to determining when values of pure functional languages can be modified destructively rather than by copying. Guzman and Hudak [GH90] propose a typed extension of functional languages called *single threaded lambda calculus* that can express the sequencing constraints required for the in-place update of array-like data structures. Wadler [Wad90b] proposes a similar solution using types motivated by Girard's Linear Logic and, later, shows the two approaches to be equivalent [Wad91]. He also proposes an alternate solution inspired by monad comprehensions [Wad90a].

These approaches differ radically from ours in that they do not treat references as values. Programming is still done in the functional style (that is, using our τ types). Shared updates cannot be expressed, and *pointers* (references to references) and *objects* (mutable data structures with function components) are absent. Although it is possible to represent references as indices into an array called the store, the result is a low-level "Fortran-style" of programming, and it is not apparent how references of *different types* can be accommodated.

Continuation-based effects Our approach to incorporating state changes is closely related to (and inspired by) continuation-based input/output methods used in functional languages [HW90, Kar81, Per90, MH]. The early proposal of Haskell incorporated continuation-based I/O as a primitive mechanism, but Haskell version 1.0 defines it in terms of stream-based I/O [HW90, HS88]. Our `Obs` types are a generalization of the Haskell type `Dialog`. In ILC, `Dialog` can be defined as `Obs Unit` where `Unit` is a one-element type.

Effect systems An effect system of Gifford and Lucassen [GL86] is a type system that describes the side-effects that expressions can have. A compiler can then use this information to determine when expressions can be evaluated in parallel, or when they may be memoized without altering the meaning of the program. The side-effect information computed by Gifford and Lucassen assumes an eager order of evaluation; this contrasts with our goal of handling assignments in lazy languages.

Equational axiomatizations Felleisen [Fel88, FF87, FH89], Mason and Talcott [MT89a, MT89b] give equational calculi for untyped Scheme-like languages with side effects. The calculi are based on the notion of *observational equivalence*: two terms are equivalent if they yield the same result in all contexts of atomic type. Our reduction system bears some degree of similarity to these calculi. However, the calculi are considerably more complex than our reduction system because of the possibility of side effects. We are investigating the formal relationships between the different approaches.

Laws of programming In a recent paper, Hoare et. al. [HHJ+87] present an equational calculus for a simple imperative language without procedures. The equations can be oriented as reduction rules and used to normalize recursion-free command phrases. Our work is inspired, in part, by this equational calculus.

Algol-like languages In a series of papers [Rey81, Rey82], Reynolds describes a language framework called *Idealized Algol* which is later developed into the programming language *Forsythe* [Rey88]. Forsythe has a two-layered operational semantics: the reduction semantics of the typed lambda calculus, and a state transition semantics. The former expands procedure calls to (potentially infinite) normal forms, while the latter executes the commands that occur in the normal forms. Forsythe is based on the principle that the lambda calculus layer is independent of the state transition layer. In particular, references to functions are not permitted because assignments to such references would affect β-expansion.

In contrast, our operational semantics involves a single *unified* reduction system that includes both β-expansion and command execution. Therefore, Forsythe's restrictions do not appear in our formulation. At the level of terms, ILC contains an applicative sublanguage (in terms of τ types) which is absent in Forsythe. Further, ILC permits state-independent imperative terms to be coerced to applicative types thereby allowing functions that create and use local state. No similar coercion is available in Forsythe.

8 Conclusion

We have presented a formal basis for adding mutable references and assignments to applicative languages without violating the principle of referential transparency. This is achieved through a rich type system that distinguishes between state-dependent and state-independent expressions and sequentializes modifications to the state. The language possesses the desired properties of applicative languages such as strong normalization and confluence. At the same time, it allows the efficient encoding of state-oriented algorithms and linked data structures.

integration of functional and imperative programming paradigms. Their differing strengths are orthogonal, but not conflicting. Much further work remains to be done regarding the approach presented here. The issues of polymorphism over mutable and observer types must be investigated. A complete equational calculus must be found for supporting formal reasoning. This, in turn, requires a formalization of the models of ILC and the development of proof methods like logical relations. The incorporation of an effect system and use of the monad comprehension notation would make the language more flexible and convenient to use. Finally, the issues of implementation need to be addressed.

Acknowledgements We thank John Gray, Jim Hook, Matthias Felleisen, Sam Kamin, Dave MacQueen, Ian Mason, John Ramsdell, John Reynolds, Peter Sestoft, Harald Sondergard, Carolyn Talcott, Satish Thatte and Phil Wadler for numerous discussions which led to vast improvements in our presentation.

References

[Fel88] M. Felleisen. lambda-v-cs: An extended lambda-calculus for scheme. In *ACM Symp. on LISP and Functional Programming*, 1988.

[FF87] M. Felleisen and D. P. Friedman. A calculus for assignments in higher-order languages. In *ACM Symp. on Principles of Programming Languages*, pages 314–325, 1987.

[FH89] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. Technical Report COMP TR89-100, Rice University, 1989.

[GH90] J.C. Guzman and P. Hudak. Single-threaded polymorphic lambda calculus. In *IEEE Symp. on Logic in Computer Science*, pages 333–343, 1990.

[GL86] D.K. Gifford and J.M. Lucassen. Integrating functional and imperative programming. In *ACM Symp. on LISP and Functional Programming*, pages 28–38, 1986.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[HB85] P. Hudak and A. Bloss. The aggregate update problem in functional programming systems. In *ACM Symp. on Principles of Programming Languages*, pages 300–314, 1985.

[HHJ+87] C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Communications of the ACM*, 30(8):672–686, August 1987.

[HS88] P. Hudak and R. Sundaresh. On the expressiveness of purely functional I/O systems. Technical Report YALEU/DCS/RR665, Yale University, Dec 1988.

[Hue80] G. Huet. Confluent reductions: abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, October 1980. (Previous version in *Proc. Symp. Foundations of Computer Science*, Oct 1977).

[Hug90] J. Hughes. Why functional programming matters. In *Research Topics in Functional Programming*, Univ. of Texas at Austin Year of Programming Series, chapter 2, pages 17–42. Addison-Wesley, 1990.

[HW90] P. Hudak and P. Wadler (editors). Report on the programming language Haskell, A non-strict purely functional language (Version 1.0). Technical Report YALEU/DCS/RR-777, Dep. of Computer Sc., Yale University, Apr 1990.

[Kar81] K. Karlsson. Nebula, A functional operating system. Tech. report, Chalmers University, 1981.

[LG88] J.M. Lucassen and D.K. Gifford. Polymorphic effect systems. In *ACM Symp. on Principles of Programming Languages*, pages 47–57, 1988.

[MH] L. M. McLoughlin and S. Hayes. Interlanguage working from a pure functional language. Functional Programming mailing list, Nov 1988.

[Mit90] J. C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. North-Holland, Amsterdam, 1990. (also Report No. STAN-CS-89-1277, Department of Computer Science, Stanford University).

[MT89a] I. A. Mason and C. Talcott. Axiomatizing operational equivalence in the presence of side effects. In *IEEE Symp. on Logic in Computer Science*, pages 284–293. IEEE, 1989.

[MT89b] I. A. Mason and C. Talcott. A sound and complete axiomatization of operational equivalence between programs with memory. Technical Report STAN-CS-89-1250, Stanford University, 1989. (to appear in *Theoretical Computer Science*).

[MTH90] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. The MIT Press, Cambridge, Massachusetts, 1990.

[Per90] N. Perry. *The Implementation of Practical Functional Programming Languages*. PhD thesis, Imperial College of Science, Technology and Medicine, University of London, 1990.

[Pra71] D. Prawitz. Ideas and results in proof theory. In *Proc. Second Scandinavian Logic Symposium*, 1971.

[RC86] J. Rees and W. Clinger (editors). Revised^3 report on the algorithmic language scheme. *ACM SIGPLAN Notices*, 21(12):37–79, Dec 1986.

[Rey81] J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. North-Holland, 1981.

[Rey82] J. C. Reynolds. Idealized Algol and its specification logic. In Neel. D., editor, *Tools and Notions for Program Construction*, pages 121–161. Cambridge Univ. Press, 1982.

[Rey88] J.C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, June 1988.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[Sch85] D. A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, Apr 1985.

[SR91] V. Swarup and U.S. Reddy. A logical view of assignments. In *Conf. on Constructivity in Computer Science*, 1991. (To appear).

[Sto77] J. E. Stoy. *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[SW74] C. Strachey and C. P. Wadsworth. Continuations - a mathematical semantics for handling full jumps. Tech. Monograph PRG-11, Programming Research Group, University of Oxford, 1974.

[Swa91] V. Swarup. *Type theoretic properties of assignments*. PhD thesis, University of Illinois at Urbana-Champaign, 1991. (To appear).

[Tai75] W. W. Tait. A realizability interpretation of the theory of species. In R. Parikh, editor, *Proceedings of Logic Colloquium*, volume 453 of *Lecture Notes in Mathematics*, pages 240–251. Springer, Berlin, 1975.

[Tof88] M. Tofte. *Operational semantics and polymorphic type inference*. PhD thesis, Edinburgh University, 1988. Available as Edinburgh Univ. Lab. for Foundations of Computer Science Technical Report ECS-LFCS-88-54.

[Wad90a] P. Wadler. Comprehending monads. In *ACM Symp. on LISP and Functional Programming*, 1990.

[Wad90b] P. Wadler. Linear types can change the world. In *IFIP Working Conf. on Programming Concepts and Methods*, Sea of Gallilee, Israel, Apr 1990.

[Wad91] P. Wadler. Is there a use for linear logic? In *Proc. ACM SIGPLAN Conf. on Partial Evaluation and Semantics-Based Program Manipulation*, New York, 1991. ACM. (SIGPLAN Notices, to appear).

In Proceedings of the 6th Annual Knowledge-Based Software Engineering Conference, September 22-25, 1991, Syracuse, NY.

53-61
94022
12p.
245145

Acquiring Software Design Schemas: A Machine Learning Perspective

Mehdi T. Harandi and Hing-Yan Lee*
Department of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield Avenue, Urbana, IL 61801
harandi@cs.uiuc.edu hingyan@cs.uiuc.edu

Abstract

In this paper, we describe an approach based on machine learning that acquires software design schemas from design cases of existing applications. An overview of the technique, design representation, and acquisition system, is presented. The paper also addresses issues associated with generalizing common features such as biases. The generalization process is illustrated using an example.

1 Introduction

Research in software reuse has hitherto focused on how software, be it program code, design, or specification, can be applied in similar subsequent situations. Although the notion of reusing software abstractions (also referred to as schemas, templates, and clichés) is attractive for many reasons, including greater reuse potential, ease of understanding, and reduced complexity, it remains a dream unfulfilled. The main obstacle to the practical use of schema-based systems (e.g., [3,4]) lies in the acquisition of schemas. Currently schemas have to be laboriously handcrafted.

Our research seeks to address this problem in the context of a schema-based design system, IDeA [10,11]. Our computational model is centered upon a bottom-up case-based approach [17] to be used by domain analysts. Design cases from existing applications for a design family are examined and analyzed to identify similar features (i.e., objects and operations) across examples.

***Supported in part by a National Computer Board (Singapore) postgraduate scholarship.**

2 A Machine Learning Approach

Machine learning research is concerned with computational theories of learning and development of learning tools [12]. In particular, similarity-based learning (SBL) techniques [8] provide an attractive tool for supporting the acquisition of design schemas. SBL algorithms operate by comparing externally supplied training examples to find similarities and then produce general rules or procedures from these common features [15]. We adopt a learning technique that uses positive training examples with bounded generalization.

2.1 Acquiring and Improving Schemas

To derive a design schema from design cases, mappings between corresponding features must be established. The object correspondences are obtained using domain knowledge. Where this is unavailable, analogies between objects, detected using background knowledge, are used. Operation correspondences are established using domain knowledge, operation structure knowledge and a similarity measure. Both syntactic and semantic matchings are used. In matching objects, object type knowledge is used where available; otherwise a similarity score based on data types of the attributes of objects is computed to determine the degree of match. The mapping process produces a set of common features that is then abstracted. This abstraction process is the subject matter of this paper.

Automated schema acquisition provides a solution to the problem of initially populating a design schema library. A schema, however, can be improved by refining it with new design cases encountered subsequently, removing any arbitrariness. Depending on the number of available design cases, different strategies can be adopted. One may opt for an *eager* abstraction strategy, deriving a schema whenever two cases are available. Or one may defer schema derivation for as long as possible (the *lazy* abstraction strategy), retaining cases as they are obtained and deriving a schema only when it is needed for design. Each strategy has its own merits and shortcomings. The former has a conservative storage policy, anxious to discard cases for a design family after its schema is derived. However, insufficient cases would have been examined to detect frequency of design pattern occurrences that are useful to schema derivation. The latter strategy is inherently a storage hog. Because schema derivation requires interaction with a domain analyst, attempting to perform derivation when the schema is needed is impractical. The trade-off is likely to lie somewhere in between, possibly near the eager abstraction end i.e., $1 \leq$ case retention threshold $\ll n$ where n is a large number. The primary implication of this for a pure schema-based design approach is that cases co-exist with schemas in the design library, resulting in a hybrid approach to software design.

The acquisition/improvement algorithm is sketched in Figure 1. Since there are in general more cases than schemas, schema retrieval is attempted first. The retrieved schemas

```
if (retrieve(NewDesignCase, designSchema) == OK) and (one is chosen)
   then improve-schema
   else if (retrieve(NewDesignCase, designCase) == OK) and (one is chosen)
           then if (case retention threshold exceeded)
                   then derive-schema
                   else store(NewDesignCase)
           else store(NewDesignCase)
```

Figure 1: Top-Level Schema Acquisition/Improvement Algorithm

are ranked and presented to the user to choose an appropriate one, which is then refined using the new design case. When no schema is retrieved or is found suitable, case retrieval is attempted. The retrieved cases are ranked and the one belonging to the same design family as the new design case is then chosen. If the case retention threshold is exceeded, the new design case is used together with other cases for that design family to derive a schema; otherwise it is stored along with other cases for the same design family. If case retrieval fails or no case is deemed suitable, a domain dictionary is created for the new design case (this assumes its design family is not known to the system yet).

2.2 Design Representation

The input to the acquisition system consists of one or more design cases. Each design case consists of two components: a data component and a process component. The data component is represented by an entity-relationship (ER) data model [5], where the entity types correspond to objects. The process component is represented by a set of layered dataflow diagrams (DFDs) [7], in which the transformations correspond to operations. In DFDs, the data stores correspond to the entity types and the dataflows are associated with data types. The representation of cases is the same as schemas, different only in their level of abstraction. Semantic information is associated with ER models and DFDs. For example, entity types are either regular or weak, while attributes have properties such as attribute kind (basic, acquired, or derived), attribute type (key or non-key), and attribute structure (simple or composite).

2.3 System Overview

Figure 2 shows a schematic overview of a design environment incorporating the schema acquisition/improvement subsystem, a knowledge base, an analogy-based retriever and the

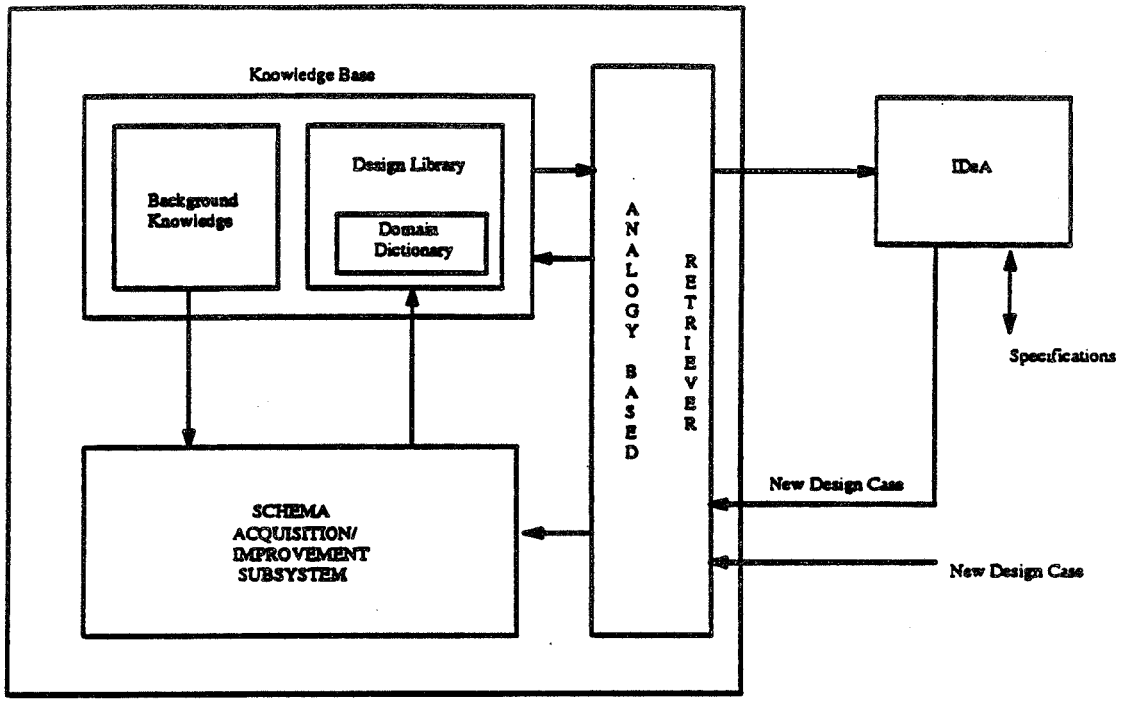


Figure 2: System Overview

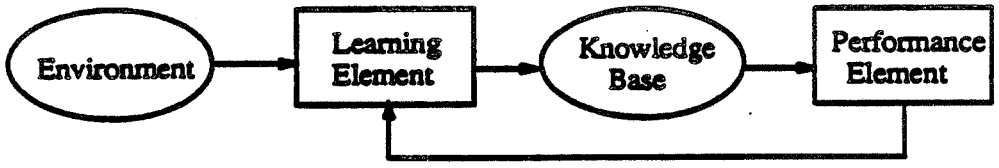


Figure 3: Machine Learning Model

IDeA design system. The knowledge base has two components: background knowledge in the form of an object type lattice and a data type lattice, and a design library containing domain-specific schemas and their associated refinement and specialization rules. The analogy-based retriever supports both design and schema acquisition/improvement. In the design mode, the user-specified objects are used to fetch schemas and cases using the object type lattice. If this fails, an object similarity-based approach is used whereby attributes of the objects are used to determine the degree of similarity between user-specified objects and those in the object type lattice. In the acquisition/improvement mode, objects from the new case are used in a similar way.

The above system fits the machine learning model (reproduced in Figure 3) discussed in [6], where IDeA and the software designer correspond to the performance element and the schema acquisition/improvement subsystem to the learning element, and the domain analyst plays the role of the environment.

3 Example

Figures 4 and 5 show the process components of design cases for a book library and a video library, respectively. A schema derived from these two designs is depicted in Figure 6. The inset in each figure is a decomposition tree, showing the operations at various levels of abstraction; the root of each tree corresponds to the top-level DFD. The lack of space prevents the inclusion of data models associated with the cases. We will use this example to illustrate several aspects in operation abstraction of schema derivation. These include dropping of non common features, detection of missing abstraction level, detection of overlapping abstraction level, handling of dataflow orphans, and handling of dataflow discontinuities.

4 Abstraction

Abstraction involves generalization, aggregation, classification, and identification [14,16]. The desired result is a succinct description that hides details and focuses on the common features. Design abstractions are useful for helping to manage the intellectual complexity that software design entails. Classification is concerned with how the schemas for design families and subfamilies are organized. As design schemas are added or refined, the respective family and subfamily relationships are modified accordingly. Identification involves providing a system of indices to the objects and operations to facilitate the location of appropriate schemas in the design library. Aggregation refers to associating an aggregate feature with a relationship between features while generalization is concerned with regard-

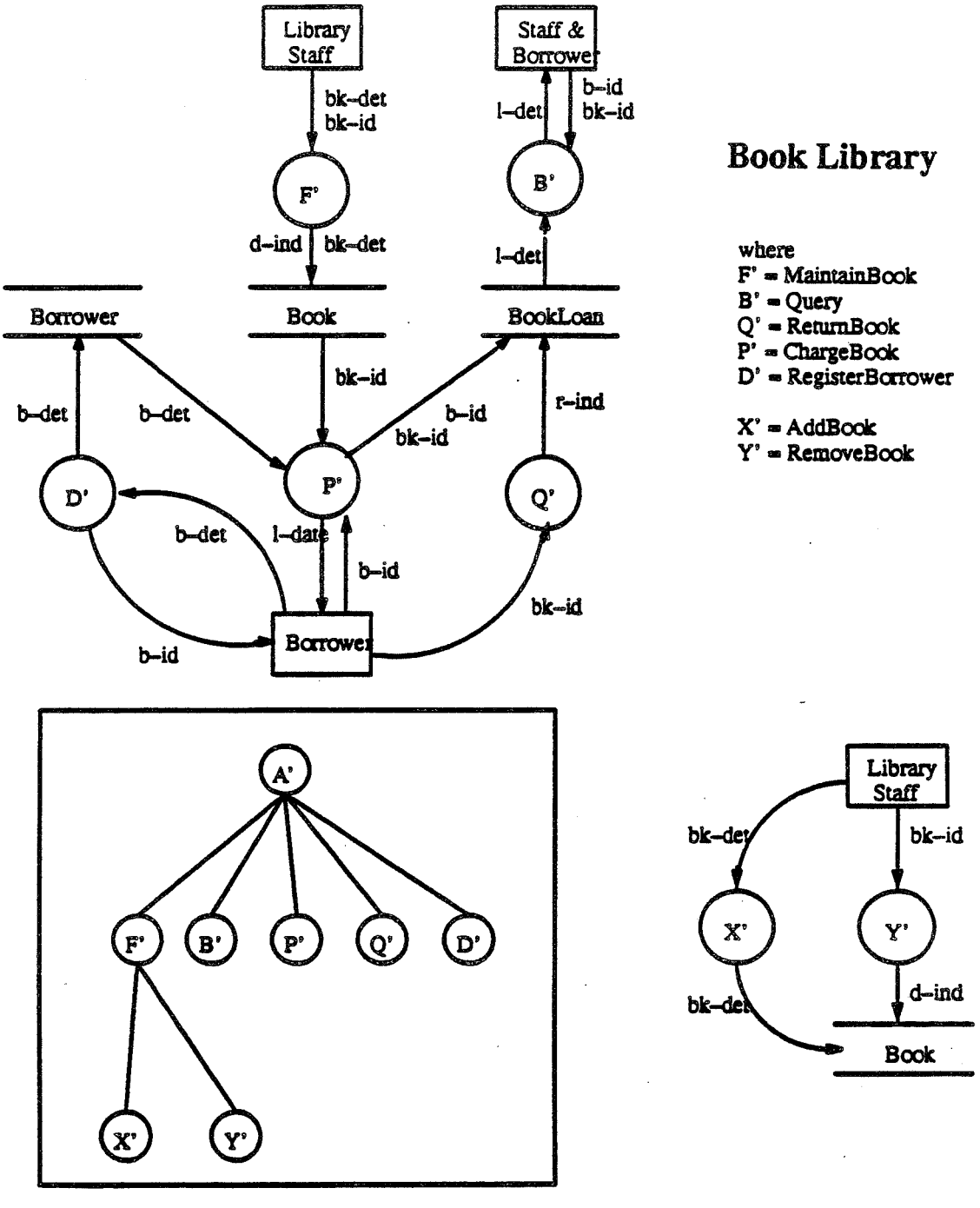


Figure 4: Book Library

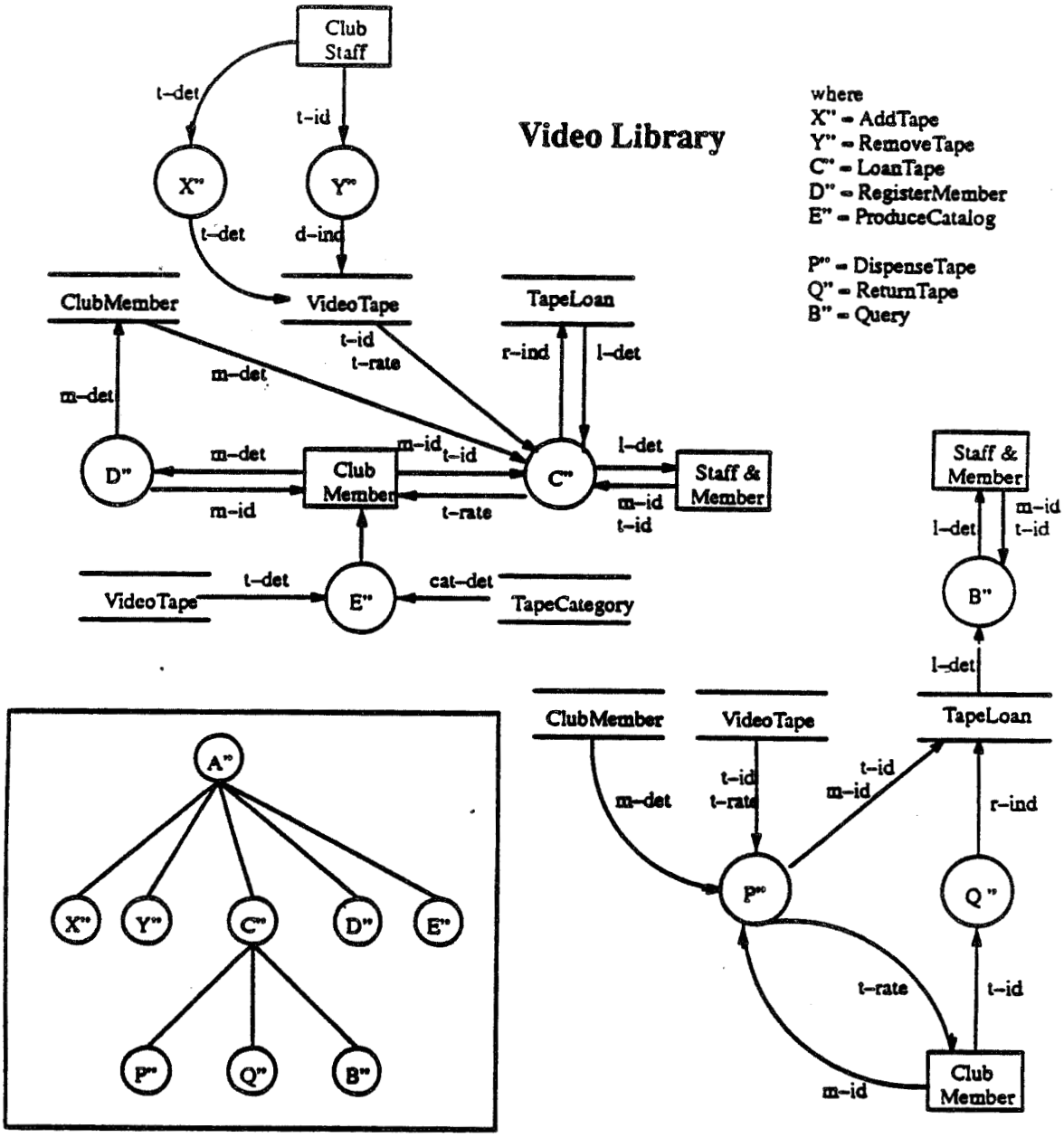


Figure 5: Video Library

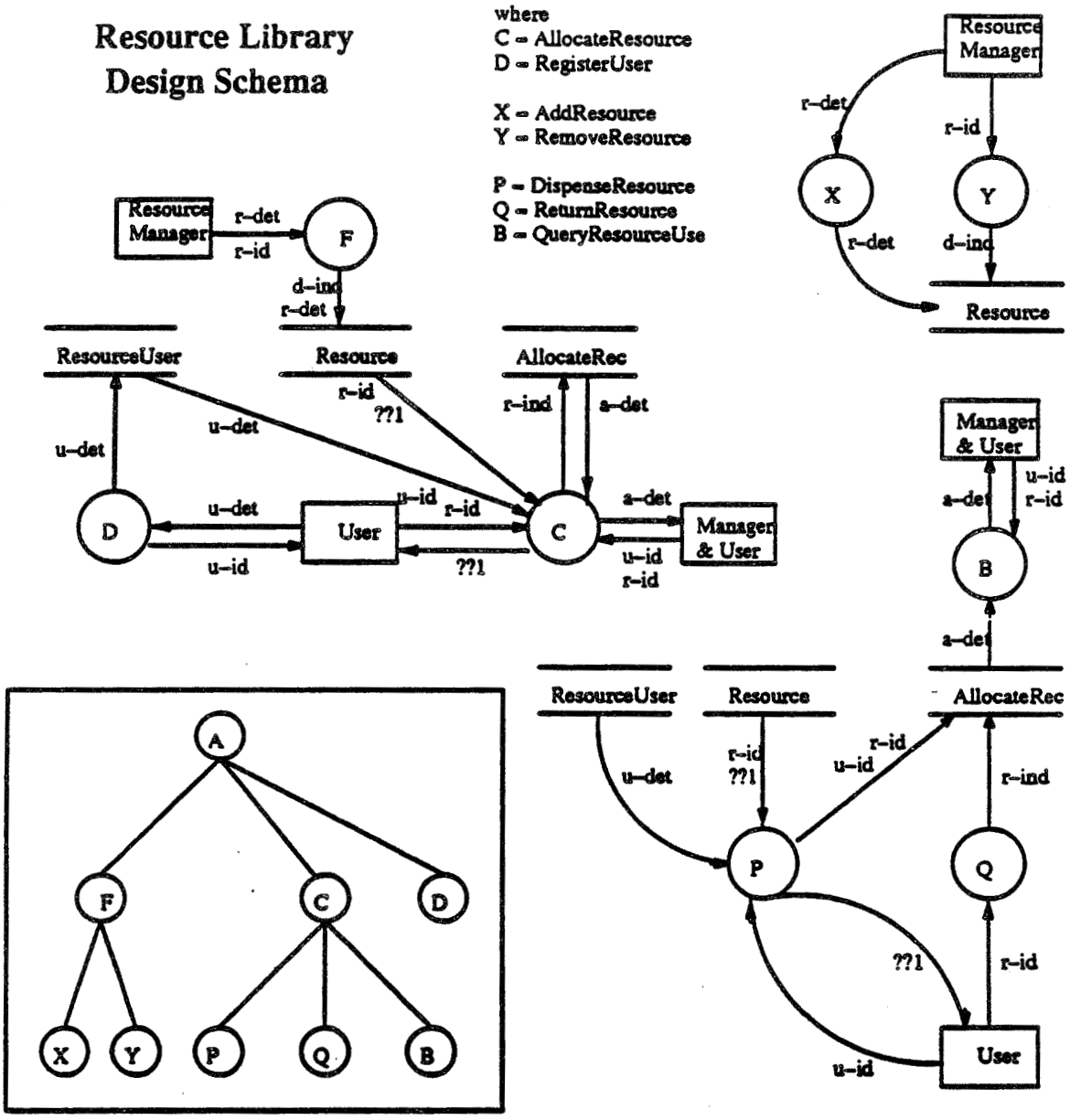


Figure 6: Resource Library Design Schema

ing a set of similar features as a generic concept. We discuss only generalization using the library cases introduced earlier.

4.1 Non Common Features

Only the essential features of a design are embodied in a design schema. Essential features are taken to be those features common to the cases encountered so far. Although the frequency of feature occurrence is used as an indication of importance, this is a heuristic approximation. This measure is not perfect since some commonly perceived features may actually be spurious [2]. The ProduceCatalog operation from the video library case has no correspondence in the book library case and is excluded from the schema.

4.2 Adoption of Biases

For cases in a design family, there may be several possible corresponding design schemas. It is hence necessary to introduce *biases* that help determine which schema to prefer. Without the bias, there is no basis to favor one schema over another [13].

4.2.1 Missing Abstraction Level

The leaf nodes of a decomposition represent primitive operations (or P-ops) while the interior nodes represent the abstract operations (or A-ops). An A-op at a given level constitutes an abstraction level with respect to those operations it is decomposed into at the next lower level. A missing abstraction level hence refers to the absence of an A-op in one design case, for which mappings can be found for its constituent operations. The generalization process needs to recognize and detect missing abstraction levels that occur across cases. For example, there is no A-op in the video library case that maps to MaintainBook in the book library case, although the latter's constitutent P-ops, AddBook and RemoveBook, have mappings to AddTape and RemoveTape. Similarly, LoanTape in the video library case has no corresponding A-op in the book library case.

To reconcile this discrepency in the design schema, two alternatives exist; excluding the missing abstraction level or including it. The bias we use is to include the missing A-op. The application of this bias means that the resultant decomposition tree is no taller than the tallest of the input cases. In our example, the library schema incorporates both the unmapped (missing) A-ops, MaintainBook and LoanTape.

4.2.2 Overlapping Abstraction Level

When an operation in one case appears at an abstraction level that is different from the corresponding operation in another case, an overlapping abstraction level is found. This

situation arises for Query (B') in the book library case. Although it has a correspondence in Query (B") in the video library case, they appear at different abstraction levels. Since there appears no real justification to prefer either in the derived schema, we adopt the bias that favors the lower (with respect to the level in the decomposition tree) of the two operations in the schema. In our example, QueryResourceUse (B) in the derived schema is placed in the same relative position as Query (B").

4.3 Abstracting Dataflows

At each abstraction level in a schema, the matching operations and dataflows are generalized. Dataflow generalization is an informed one in that it is guided by object and attribute mappings. It can be accomplished in a straightforward manner, when due attention is paid to dataflow orphans and discontinuities.

4.3.1 Dataflow Orphans

A dataflow orphan is a dataflow in a schema that is generalized from two corresponding dataflows with non-matching data types. In our example, a dataflow orphan (labelled ??1) in the library schema is created as a result of an outgoing dataflow from DispenseResource to User. This dataflow arises because the date_due dataflow from the book library and the rental_rate dataflow from the video library do not have matching data types.

4.3.2 Dataflow Discontinuities

Dataflows for an operation in a schema may suffer from discontinuity when its neighbor operations are not generalized and included. The abstract operation for PrintReport2 (p3 and p3') in Figure 7 has a dataflow discontinuity, because the non common SORT operation in (b) is excluded. The generalization process must recognize that the incoming dataflow is from DS.

5 Related Work

The problem of establishing feature correspondence between cases is similar to that encountered in view integration for database design. While previous methods use inter-view and intra-view assertions, our approach uses domain knowledge, heuristics and similarity measures [9].

The planning system of Anderson and Farley [1], PLANEREUS, creates abstract operators while organizing a given set of primitive ABSTRIPS-type operators into a taxonomic

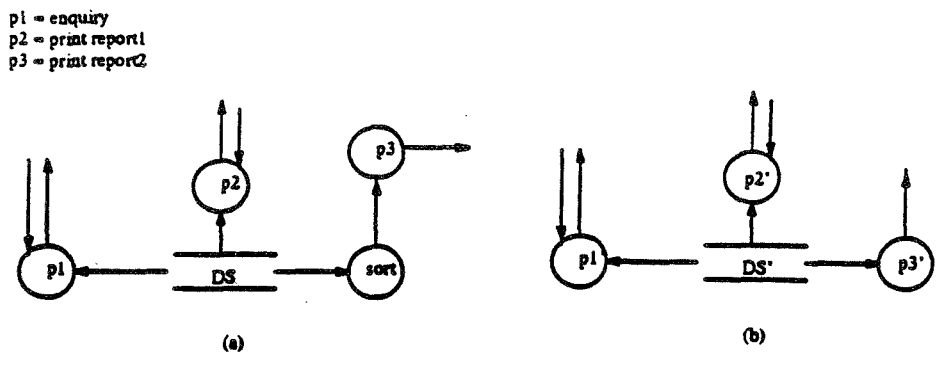


Figure 7: Dataflow Discontinuity

hierarchy and categories of abstract object types. While our goal is similar to theirs, the details differ. The ABSTRIPS-type operators do not have the layered structure DFDs have. PLANEREUS matches operators first, followed by objects. In our domain, knowledge of objects and operations is much richer. We map objects based on available object type knowledge, attributes and structural knowledge. These object mappings then facilitate operations mapping.

In SBL, syntactic rules of inference are used to accomplish generalization in predicate calculus. These inductive rules include turning constants to variables, dropping conditions, and adding options [6]. These rules have parallels in our generalization process. The exclusion of non common features from a schema constitutes the dropping conditions rule. The turning of constants to variables rule is analogous to a schema matching more cases than those from which it has been derived. Adding options occurs in the optional dataflows that are present in most but not all the input cases.

6 Summary

A solution to the problem of acquiring schemas for a schema-based software design system is presented. We described a bottom-up case-based schema derivation approach that is amenable to automation using AI techniques. This machine learning attempt at schema acquisition is but a first step toward overcoming the problem of library population in a reusable environment. We also showed how schemas should co-exist with cases in a schema-based system, when it is considered prudent to defer schema derivation until an adequate number of cases can be obtained.

References

[1] John S. Anderson and Arthur M. Farley. Plan Abstraction Based on Operator Generalization. In *Proc. AAAI Natl. Conf. on Artificial Intelligence*, pages 100 – 104, St. Paul, MN, August 1988.

[2] Ray E. Bareiss. Protos: An Exemplar-Based Learning Apprentice. In *Proc. 4th Intl. Workshop on Machine Learning*, pages 12 – 23, Morgan Kaufmann Publishers, Irvine, CA, June 1987.

[3] Grady H. Campbell Jr. *Abstraction-based Reuse Repositories.* Technical Report 89041-N, Software Productivity Consortium, Inc., Hendron, VA, July 1989.

[4] Tom E. Cheatham, G.H. Holloway, and J.A. Tomley. Program Refinement. In *Proc. 5th Intl. Conf. on Software Engineering*, pages 430 – 437, San Diego, CA, 1979.

[5] Peter P.S. Chen. The Entity Relationship Model: Towards a Unified View of Data. *ACM Trans. on Database Systems*, 1(1):9 – 36, 1976.

[6] Paul R. Cohen and Edward A. Feigenbaum, editors. *The Handbook of Artificial Intelligence.* Volume 2, Addison-Wesley, 1982.

[7] Tom DeMarco. *Structured Analysis and System Specification.* Yourdon, Inc., New York, NY, 1979.

[8] Thomas G. Dietterich and Ryszard S. Michalski. A Comparative Review of Selected Methods of Learning from Examples. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, Morgan Kaufmann Publishers, Palo Alto, CA, 1983.

[9] Hing-Yan Lee and Mehdi T. Harandi. A Knowledge-based Approach to Correspondence Identification and Conflict Detection in View Integration. In *Proc. 4th Intl. Symposium on Artificial Intelligence*, Cancun, Mexico, November 1991. (to appear).

[10] Mitchell D. Lubars. The IDeA Design Environment. In *Proc. 11th Intl. Conf. on Software Engineering*, pages 23 – 32, Pittsburgh, PA, May 1989.

[11] Mitchell D. Lubars and Mehdi T. Harandi. Knowledge-Based Software Design Using Design Schemas. In *Proc. 9th Intl. Conf. on Software Engineering*, pages 253 – 262, Monterey, CA, March 1987.

[12] Ryszard S. Michalski. Understanding the Nature of Learning: Issues and Research Directions. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning II*, pages 311 – 348, Morgan Kaufmann Publishers, Palo Alto, CA, 1987.

[13] Tom M. Mitchell. *The Need for Biases in Learning Generalizations.* Technical Report CBM-TR-117, Dept. of Computer Science, Rutgers University, New Brunswick, NJ, May 1980.

[14] John Mylopoulus. An Overview of Knowledge Representation. *Proc. Workshop on Data Abstraction, Databases and Conceptual Modeling, ACM SIGMOD Record*, 11(2):5 – 14, January 1981.

[15] Jude W. Shavlik and Thomas G. Dietterich, editors. *Readings in Machine Learning.* Morgan Kaufmann Publishers, 1990.

[16] John M. Smith and Diana C.P. Smith. Database Abstraction: Aggregation and Generalization. *ACM Trans. on Database Systems*, 2(2):105 – 133, June 1977.

[17] Will Tracz. RMISE Workshop on Software Reuse Meeting Summary. In Will Tracz, editor, *Tutorial: Software Reuse: Emerging Technology*, pages 41 – 53, IEEE Computer Society, 1988.

Workshop Notes from the Ninth
National Conference on
Artificial Intelligence

(AAAI-91)

Automating Software Design: Interactive Design

American Association for Artificial Intelligence

July, 15, 1991

Anaheim, California

Some Interactive Aspects of a Software Design Schema Acquisition Tool

514-61
94023 6p.
24514

Hing-Yan Lee and Mehdi T. Harandi
Department of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield Avenue, Urbana, IL 61801
hingyan@cs.uiuc.edu harandi@cs.uiuc.edu

Abstract

This paper describes a design schema acquisition tool which forms an important component of a hybrid software design system for reuse. The hybrid system incorporates both schema-based and case-based approaches in supporting software design reuse activities and is realised by extensions to the IDeA system [Lubars, 1989, Lubars and Harandi, 1986]. The paper also examines some of the interactive aspects that the tool requires with the domain analyst to accomplish its acquisition task.

Introduction

A *schema* is an abstraction or abstract representation of some artifact or concept; it has also been referred to by other names, including plans, clichés and templates in the automatic programming literature. A design schema, therefore, is an abstraction of some design artifact.

In this paper, we describe a design schema acquisition tool, which forms an important component of a hybrid software design system for reuse. We also indicate aspects of the tool that require interaction with the domain analyst to enable it to complete the derivation of a design schema. Such interactions arise in situations where the tool is unable to resolve competing conflicts, where there are gaps in the knowledge base, and where the assignment of meaningful (i.e., human understandable) names and labels is needed.

The *hybrid software design system* is the result of the evolution of a schema-based software design system (SBD), namely IDeA. We have extended IDeA's capabilities to compensate for some of its inherent shortcomings.

A Hybrid Software Design System

We briefly describe IDeA, discuss its problems from a design schema acquisition perspective, and then detail extensions to IDeA for overcoming these weaknesses. The extended system is a hybrid software design system.

IDeA

The IDeA system [Lubars, 1989, Lubars and Harandi, 1986] is a prototypical design environment that, besides providing support for many aspects of software design, adopts a knowledge-based approach in its support for software reuse. It is based on the dataflow model of software design and is suited to support structured analysis and design [DeMarco, 1979].

IDeA has a library of design schemas [Lubars, 1986, Lubars and Harandi, 1987]. Each schema has a set of associated refinement and specialization rules. Specialization rules allow IDeA to move downwards through a pre-existing abstraction hierarchy from a particular design while refinement rules add levels of details to the dataflow design by expanding a single operation to one that consists of several lower level operations with intermediate dataflows.

IDeA supports top-down design by attempting to retrieve a design schema satisfying user-supplied high-level specifications. If necessary, it adopts a bottom-up design approach by attempting to compose a design from several design schemas.

Shortcomings of SBDs

SBDs have three major shortcomings, the solutions of which are crucial to their success. Shortcomings particular to IDeA have been described elsewhere [Lubars and Harandi, 1988].

Firstly, as abstractions, design schemas embody only the essential and common aspects of a design family. The non-common details, which may be useful in subsequent designs, are omitted.

Secondly, such systems assume a well-defined domain boundary. However, establishing the domain boundary is by no means an easy task [Lubars, 1988].

Thirdly, the issue of acquisition of design schemas has largely been unaddressed to date. It is, however, widely recognized that the process of constructing reusable design schemas constitutes a serious bottleneck for its practical application. Indeed this bottleneck, called the *schema library population problem*, is prevalent in all SBDs; this is a specialization of the classical knowledge acquisition problem in AI.

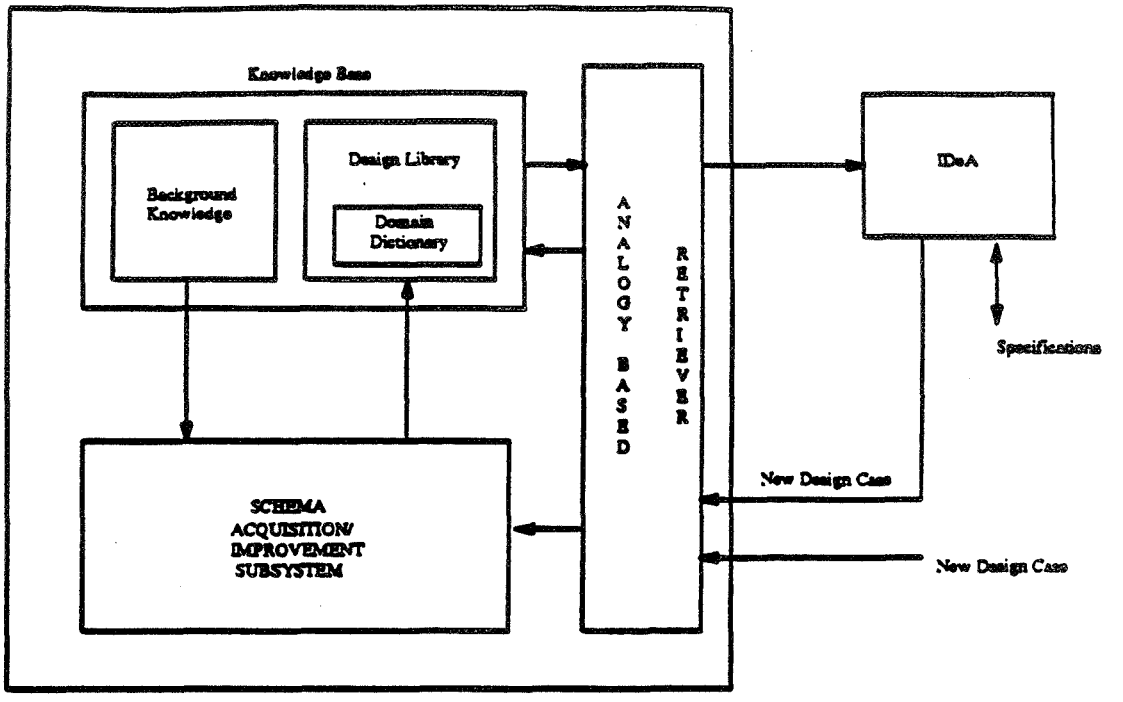


Figure 1: Overview of IDeA II

Currently, each design schema has to be painstakingly hand-crafted. This manual mode of acquisition is not only laborious, but also error-prone. To alleviate this, automated aquisition, even if it is only partial or computer-aided, is necessary. Support for later improvement and refinement of the acquired design schemas is also desirable.

Extensions to IDeA

To address problems associated with the schema library population and the subsequent design schema improvement issues, we are working on several extensions to the original IDeA system. We shall hereafter refer to the extended system as IDeA II. An overview of IDeA II is depicted in Figure 1.

Design Representation The representation of design cases is the same as design schemas, different only in their level of abstraction; each has two components:

- a data component, represented by an entity-relationship (ER) data model [Chen, 1976], where the entity types correspond to object types, entities to objects, data attributes to object attributes, entity key to object identity, and relationships to object relationships. The ER model is a semantic data model, with levels of abstraction in its representation. Aggregation (*is-part-of*) relationships exist for attributes of objects and for attributes of attributes (i.e., composite attributes). Generalisation (*isa*) relationships exist between objects.
- a process component, represented by a set of layered dataflow diagrams [DeMarco, 1979], in which the transformations correspond to operations. The data stores correspond to the objects represented by the entity types in the data models.

Schema Acquisition and Improvement The primary extension entails a design schema acquisition and improvement system which not only expediates the initial population of the schema library but also facilitates subsequent design schema improvements. The system adopts a bottom-up case-based approach [Schoen *et al.*, 1989, Tracz, 1988] (as opposed to a top-down model-based approach) to design schema acquisition. In this approach, specific design cases from existing applications are examined and analysed to identify similar objects and operations across examples. The schematic knowledge is captured using domain knowledge. Where this is unavailable, analogies between design cases, detected using background knowledge, are used to abstract the schematic knowledge. The abstraction process applies induction and similarity-based learning techniques [Dietterich and Michalski, 1983]. The design schema acquisition/improvement process has six stages:

1. mapping and abstraction of objects and attributes;

2. mapping and selection of operations;

3. validation of acquired design schema;

4. creation of refinement and specialization rules;

5. cataloging of acquired design schema and rules; and

6. cataloging of object and attribute mappings.

Stages 1 and 2 primarily involve identifying analogous design components, and performing inductive generalization.

The identification of corresponding design components is facilitated by a knowledge base that contains information on the basic object and data types as well as *isa* and *is-part-of* knowledge in the object and data type lattices. The lattices are used to determine the extent of match between objects and between attributes. The matching is based on the knowledge of objects, attributes, and their data types, rather than name string matches.

First, the correspondences between objects and between attributes in the data flow diagrams are established by examining the design cases' respective data models. This is accomplished by a data model matcher.

The mapping of objects is facilitated if their object types are known or available. This allows the matcher to directly determine whether two given objects do correspond, using the object type lattice. The extent of the match is computed from the path length between the two object types in the lattice. If the object type knowledge of the objects is unknown, then a similarity metric is used instead. The matcher compares data types of the objects' attributes, computes a similarity score for each attribute mapping and then selects the attribute mapping combination that yields the maximum total similarity score. For further details on this mechanism, see [Lee, 1991a].

Next, the object and attribute mappings as well as the abstract data model are passed to a dataflow diagram matcher and generalizer module. This module, in turn, establishes the corresponding operations and derives the required design schema. As in the data model matcher, the data type lattice forms the basis for matching. This second mapping and generalization mechanism is detailed in [Lee, 1991b].

Design schema acquisition/improvement is essentially a knowledge acquisition activity accompanied by knowledge refinement. Acquiring design schemas is easy when the system has the necessary background and domain knowledge to identify corresponding objects and operations. When this is missing, a weaker method for establishing the correspondence (i.e., using the data types of object attributes) is used and this tends to be search-intensive.

Case-based Design Deriving domain specific design schemas from design cases raises the following questions:

- How many design cases are needed (or considered adequate) to derive a design schema?
- What happens if there is only one or two design cases available?

In his work on program abstraction, Dershowitz [Dershowitz, 1985] takes the view that "using more programs reduces arbitrariness by insisting that only analogous parts be abstracted, on the presumption that they play similar roles in different programs."

A solution is to permit the co-existence of design schemas and design cases, and to treat the initial library population as a general case of schema improvement. This case-based augmentation of the schema-based approach constitutes the hybrid software design approach.

The use of design cases necessitates certain modifications to the library organization and its retrieval mechanism.

Learning Besides deriving design schemas for the design library, IDeA II improves its background knowledge by incorporating into its knowledge base newly acquired object type knowledge. For example, after the system has established that the Book object in the Book Library design case maps to the VideoTape object in the Video Library design case and that these are types of Resource, the corresponding taxonomic knowledge (i.e., Book and VideoTape's *isa* relationships with Resource) is added.

Analogy-based Retrieval Another key extension is an analogy-based retrieval mechanism for design schemas and design cases. This constrasts with the keyword method used in initial IDeA.

IDeA II first attempts to establish whether the specified objects correspond to known object types in the knowledge base. If so, it retrieves design schemas and cases that best match the rest of the specification. If the object type is unknown or match is unsuccessful, object matching based on attributes is attempted. Again design schemas and cases with the best matches are retrieved.

Interactive Aspects

In Design

The initial IDeA provides standard graphical editing capabilities for dataflow designs. The layout of the dataflow designs may be performed manually by the designer, or automatically by the system using a set of heuristics [Ning, 1984]. Multiple windows are employed to display design information, such as dataflow diagrams, dataflow transformation information, and dataflow definition. Although a natural language front-end was developed [Dahlbacka, 1985], it has not been fully integrated with IDeA.

IDeA requires interactions with the user in the following situations:

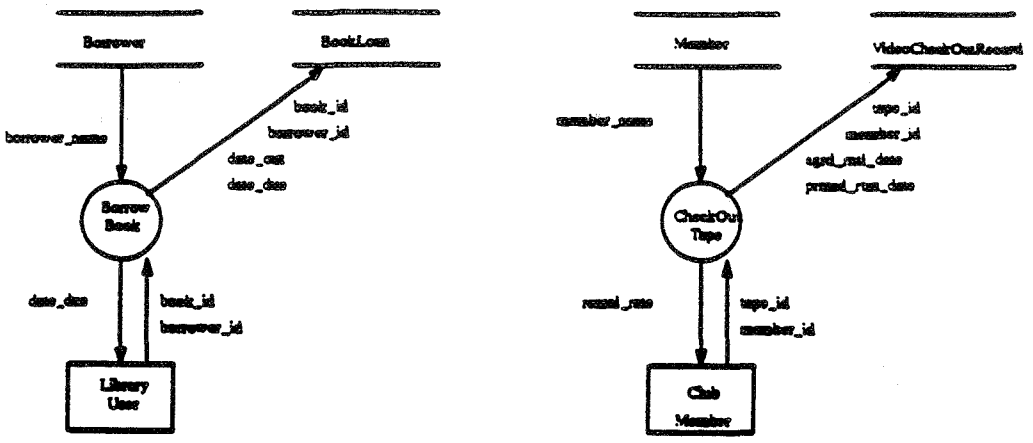


Figure 2: Operations *BorrowBook* and *CheckOutTape*

1. Whenever the system has been provided insufficient information. One situation where this can arise is during design when the high-level specification provided by the designer is insufficient for the retrieval mechanism to discern which among the several design schemas or cases satisfying the specified features is best. These are ranked and presented for selection by the designer.
2. Whenever the system has detected inconsistencies in the design. Such inconsistences may arise when the dataflows of selected design schema do not completely match those specified by the designer, because the designer-provided specifications are either incomplete or contain inaccurate information. It may also arise when a dataflow introduced at one level is not reflected at higher and/or lower levels. The interaction is achieved in a non-intrusive manner by posting the unresolved issues on a goal agenda, allowing the designer to resolve them at an opportune time.

In Schema Acquisition/Improvement

Interactions regarding schema acquisition and improvement form a major part of IDeA II's process. Interactions occur in the following cases:

1. The system has insufficient knowledge. The tool can encounter difficulties in two ways:
   - Conflict resolution. How should the system perform abstraction when two corresponding operations have incoming (or outgoing) dataflows of non-matching data types? For example, the outgoing dataflow of the BorrowBook operation to the LibraryUser external entity is the date_due which has date_type data type, while the corresponding dataflow of the CheckOutTape operation has rental_rate which has dollar_type data type (see Figure 2). Although there is no data type match, the abstract operation should reflect the outgoing dataflow (indicated by ?? in Figure 3), leaving the domain analyst to assign an appropriate (and acceptable) data type for it.
   - Match resolution. For example, in attempting to identify and establish analogical correspondence between the BookLoan object in the Book Library and the VideoCheckOutRecord object in the Video Library, there are four attributes of date_type data type (date_out, date_due date_returned and date_renewed) in the former to match against three attributes of date_type data type (agrd_rntl_date, prmsd_rtrn_date, actl_rtrn_date) in the latter. The issue here is how to determine which of these attributes correspond to each other. One solution is to specify domain constraints on attributes that would enable the system to decide the correct mappings.
2. The system seeks the domain analyst's confirmation before committing to difficult-to-reverse operations or embarking upon computationally intensive operations.
3. The system is incapable of assigning meaningful (i.e., human understandable) names and labels for abstract objects and operations in the derived design schema. Arbitrary names, such as those generated by *gensym*, are not suitable. What name should be assigned to the abstract object for the Book object in the Book Library design case and the VideoTape object in the Video Library design case? Similarly, what name should be given to the abstract operation for the BorrowBook operation from the Book

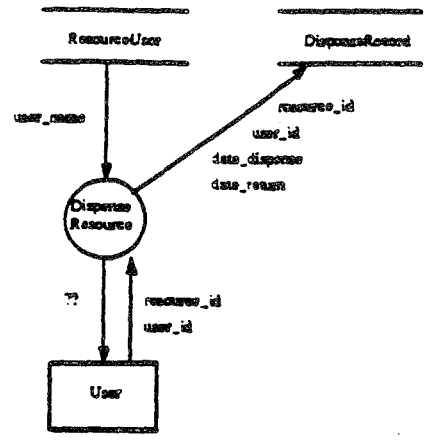


Figure 3: Abstract Operation *DispenseResource*

Library design case and the CheckOutTape operation from the Video Library design case? Possible names are indicated in Figure 3. To aid the domain analyst in assigning names, the system displays the specialized objects (or operations) for the abstract object (or operation) that is being labelled.

Summary

Design schema acquisition is important to alleviate the library population problem in schema-based systems for reuse. The subsequent refinements of these design schemas are equally important. In this paper, we have briefly described such a design schema acquisition tool and its capabilities.

However, complete automation is unlikely because this acquisition activity requires substantial amount of background and domain specific knowledge. Partial automation, augmented by interactions with the domain analyst to fill in gaps in knowledge and an acquisition tool's capabilities, is a more realistic goal.

Acknowledgements

The authors thank the two anonymous reviewers for various helpful comments. The first author is supported in part by a National Computer Board (Singapore) Postgraduate Scholarship.

References

[Chen, 1976] Peter P.S. Chen. The Entity Relationship Model: Towards a Unified View of Data. *ACM Trans. on Database Systems*, 1(1):9 – 36, 1976.

[Dahlbacka, 1985] David M. Dahlbacka. An ATN-Based Restricted Natural Language Front-end for a Dataflow Design Aid. Master's thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, 1985.

[DeMarco, 1979] Tom DeMarco. *Structured Analysis and System Specification*. Yourdon, Inc., New York NY, 1979.

[Dershowitz, 1985] Nachum Dershowitz. Program Abstraction and Instantiation. *ACM Trans. on Programming Languages and Systems*, 7(3):446 – 477, 1985. Also in Readings in Artificial Intelligence and Software Engineering, C. Rich and R. Waters, ed. Morgan Kaufmann, Los Altos, CA, 1986.

[Dietterich and Michalski, 1983] Thomas G. Dietterich and Ryszard S. Michalski. A Comparative Review of Selected Methods of Learning from Examples. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*. Morgan Kaufmann Publishers, Palo Alto, CA, 1983.

[Lee, 1991a] Hing-Yan Lee. A Knowledge-based Approach to Correspondence Identification and Conflict Detection in View Integration. Master's thesis Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, January 1991.

[Lee, 1991b] Hing-Yan Lee. Acquisition, Refinement and Retrieval of Software Design Schemas. PhD Thesis Proposal, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, April 1991.

[Lubars and Harandi, 1986] Mitchell D. Lubars and Mehdi T. Harandi. Intelligent Support for Software Specification and Design. *IEEE Expert*, 1(4):33 – 41, 1986.

[Lubars and Harandi, 1987] Mitchell D. Lubars and Mehdi T. Harandi. Knowledge-Based Software Design Using Design Schemas. In *Proc. 9th Intl. Conf. on Software Engineering*, pages 253 – 262, Monterey, CA, March 1987.

[Lubars and Harandi, 1988] Mitchell D. Lubars and Mehdi T. Harandi. The Knowledge-Based Refinement Paradigm and IDeA: Concepts, Limitations, and Future Directions. In *Proc. AAAI Workshop on Automating Software Design*, pages 68 – 73, St. Paul, MN, August 1988.

[Lubars, 1986] Mitchell D. Lubars. *A Knowledge-Based Design Aid for the Construction of Software Systems*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, November 1986. (Also UIUCDCS-R-86-1304).

[Lubars, 1988] Mitchell D. Lubars. Domain Analysis and Domain Engineering in IDeA. Technical Report STP-295-88, MCC, Austin, TX, September 1988.

[Lubars, 1989] Mitchell D. Lubars. The IDeA Design Environment. In *Proc. 11th Intl. Conf. on Software Engineering*, pages 23 – 32, Pittsburgh, PA, May 1989.

[Ning, 1984] Jim Q. Ning. Graphical Representation of Dataflow Diagrams: Design and Implementation. Master's thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, 1984.

[Schoen *et al.*, 1989] Eric Schoen, Reid G. Smith, and Bruce G. Buchanan. The Knowledge Base Designer's Assistant: A Preliminary Report. In M. Tokoro, Y. Anzai, and A.Yonezawa, editors, *Concepts and Characteristics of Knowledge-Based Systems*, pages 105 – 134. Elsevier Science, 1989.

[Tracz, 1988] Will Tracz. RMISE Workshop on Software Reuse Meeting Summary. In Will Tracz, editor, *Tutorial: Software Reuse: Emerging Technology*, pages 41 – 53. IEEE Computer Society, 1988.